Annotated Ada Reference Manual

ISO/IEC 8652:1995(E)

with Technical Corrigendum 1

and Amendment 1

Language and Standard Libraries

Consolidated Standard

Table of Contents

Foreword to this version of the Ada Reference Manual

The International Standard for the programming language Ada is ISO/IEC 8652:1995(E). 0.1/1

The Ada Working Group ISO/IEC JTC 1/SC 22/WG 9 is tasked by ISO with the work item to interpret and maintain the International Standard and to produce Technical Corrigenda, as appropriate. The technical work on the International Standard is performed by the Ada Rapporteur Group (ARG) of WG 9. In September 2000, WG 9 approved and forwarded Technical Corrigendum 1 to SC 22 for ISO approval, which was granted in February 2001. Technical Corrigendum 1 was published in June 2001. 0.2/1

In October 2002, WG 9 approved a schedule and guidelines for the preparation of an Amendment to the International Standard. WG 9 approved the scope of the Amendment in June 2004. In April 2006, WG 9 approved and forwarded the Amendment to SC 22 for approval, which was granted in August 2006. Final ISO/IEC approval is expected by early 2007. 0.3/2

The Technical Corrigendum lists the individual changes that need to be made to the text of the International Standard to correct errors, omissions or inconsistencies. The corrections specified in Technical Corrigendum 1 are part of the International Standard ISO/IEC 8652:1995(E). 0.4/1

Similarly, Amendment 1 lists the individual changes that need to be made to the text of the International Standard to add new features as well as correct errors. 0.5/2

When ISO published Technical Corrigendum 1, it did not also publish a document that merges the changes from the Technical Corrigendum into the text of the International Standard. It is not known whether ISO will publish a document that merges the changes from Technical Corrigendum and Amendment 1 into the text of the International Standard. However, ISO rules require that the project editor for the International Standard be able to produce such a document on demand. 0.6/2

This version of the Ada Reference Manual is what the project editor would provide to ISO in response to such a request. It incorporates the changes specified in the Technical Corrigendum and Amendment into the text of ISO/IEC 8652:1995(E). It should be understood that the publication of any ISO document involves changes in general format, boilerplate, headers, etc., as well as a review by professional editors that may introduce editorial changes to the text. This version of the Ada Reference Manual is therefore neither an official ISO document, nor a version guaranteed to be identical to an official ISO document, should ISO decide to reprint the International Standard incorporating an approved Technical Corrigendum and Amendment. It is nevertheless a best effort to be as close as possible to the technical content of such an updated document. In the case of a conflict between this document and Amendment 1 as approved by ISO (or between this document and Technical Corrigendum 1 in the case of paragraphs not changed by Amendment 1; or between this document and the original 8652:1995 in the case of paragraphs not changed by either Amendment 1 or Technical Corrigendum 1), the other documents contain the official text of the International Standard ISO/IEC 8652:1995(E) and its Amendment. 0.7/2

As it is very inconvenient to have the Reference Manual for Ada specified in three documents, this consolidated version of the Ada Reference Manual is made available to the public. 0.8/2

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

2 In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

3 International Standard ISO/IEC 8652 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*.

4/2 This consolidated edition updates the second edition (ISO 8652:1995).

4.a/2 **Discussion:** The above is unofficial wording added just to avoid confusion. If ISO decides to publish a new standard, the above would be replaced by “This third edition cancels and replaces the second edition (ISO 8652:1995), of which it constitutes a technical revision”. The first three paragraphs in this section also would be replaced by the current ISO boilerplate.

5/2 {*AI95-00440-01*} Annexes A to J form an integral part of this International Standard. Annexes K to Q are for information only.

5.a **Discussion:** This document is the Annotated Ada Reference Manual (AARM). It contains the entire text of the Ada 95 standard (ISO/IEC 8652:1995), plus various annotations. It is intended primarily for compiler writers, validation test writers, and other language lawyers. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

Introduction

1 This is the Annotated Ada Reference Manual.

2 Other available Ada documents include:

- 3/2 {*AI95-00387-01*} Ada 95 Rationale. This gives an introduction to the new features of Ada incorporated in the 1995 edition of this Standard, and explains the rationale behind them. Programmers unfamiliar with Ada 95 should read this first.
- 3.1/2 {*AI95-00387-01*} Ada 2005 Rationale. This gives an introduction to the changes and new features in Ada 2005 (compared with the 1995 edition), and explains the rationale behind them. Programmers should read this rationale before reading this Standard in depth.
- 4/1 *This paragraph was deleted.*
- 5 The Ada Reference Manual (RM). This is the International Standard — ISO/IEC 8652:1995.
- 5.1/2 Technical Corrigendum 1 — ISO/IEC 8652:1995:COR.1:2001. This document lists corrections to the International Standard.
- 5.2/2 Amendment 1 — ISO/IEC 8652:1995:AMD.1:2007. This document outlines additional features and corrections to the International Standard.
- 5.3/2 The consolidated Ada Reference Manual. An *unofficial* document combining the above three documents into a single document.

Design Goals

6/2 {*AI95-00387-01*} Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. The 1995 revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on reliability, maintainability, and efficiency. This amended version provides further flexibility and adds more standardized packages within the framework provided by the 1995 revision.

7 The need for languages that promote reliability and simplify maintenance is well established. Hence emphasis was placed on program readability over ease of writing. For example, the rules of the language require that program variables be explicitly declared and that their type be specified. Since the type of a variable is invariant, compilers can ensure that operations on variables are compatible with the properties intended for objects of the type. Furthermore, error-prone notations have been avoided, and the syntax of the language avoids the use of encoded forms in favor of more English-like constructs. Finally, the language offers support for separate compilation of program units in a way that facilitates program development and maintenance, and which provides the same degree of checking between units as within a unit.

8 Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep to a relatively small number of underlying concepts integrated in a consistent and systematic way while continuing to avoid the pitfalls of excessive involution. The design especially aims to provide language constructs that correspond intuitively to the normal expectations of users.

9 Like many other human activities, the development of programs is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components continues to be a central idea in the design. The concepts of packages, of private types, and of generic units are directly related to this idea, which has ramifications in many other aspects of the language. An allied concern is the maintenance of programs to match changing requirements; type

extension and the hierarchical library enable a program to be modified while minimizing disturbance to existing tested and trusted components.

10 No language can avoid the problem of efficiency. Languages that require over-elaborate compilers, or that lead to the inefficient use of storage or execution time, force these inefficiencies on all machines and on all programs. Every construct of the language was examined in the light of present implementation techniques. Any proposed construct whose implementation was unclear or that required excessive machine resources was rejected.

Language Summary

11 An Ada program is composed of one or more program units. Program units may be subprograms (which define executable algorithms), packages (which define collections of entities), task units (which define concurrent computations), protected units (which define operations for the coordinated sharing of data between tasks), or generic units (which define parameterized forms of packages and subprograms). Each program unit normally consists of two parts: a specification, containing the information that must be visible to other units, and a body, containing the implementation details, which need not be visible to other units. Most program units can be compiled separately.

12 This distinction of the specification and body, and the ability to compile units separately, allows a program to be designed, written, and tested as a set of largely independent software components.

13 An Ada program will normally make use of a library of program units of general utility. The language provides means whereby individual organizations can construct their own libraries. All libraries are structured in a hierarchical manner; this enables the logical decomposition of a subsystem into individual components. The text of a separately compiled program unit must name the library units it requires.

14 *Program Units*

15 A subprogram is the basic unit for expressing an algorithm. There are two kinds of subprograms: procedures and functions. A procedure is the means of invoking a series of actions. For example, it may read data, update variables, or produce some output. It may have parameters, to provide a controlled means of passing information between the procedure and the point of call. A function is the means of invoking the computation of a value. It is similar to a procedure, but in addition will return a result.

16 A package is the basic unit for defining a collection of logically related entities. For example, a package can be used to define a set of type declarations and associated operations. Portions of a package can be hidden from the user, thus allowing access only to the logical properties expressed by the package specification.

17 Subprogram and package units may be compiled separately and arranged in hierarchies of parent and child units giving fine control over visibility of the logical properties and their detailed implementation.

18 A task unit is the basic unit for defining a task whose sequence of actions may be executed concurrently with those of other tasks. Such tasks may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. A task unit may define either a single executing task or a task type permitting the creation of any number of similar tasks.

19/2 {*AI95-00114-01*} A protected unit is the basic unit for defining protected operations for the coordinated use of data shared between tasks. Simple mutual exclusion is provided automatically, and more elaborate sharing protocols can be defined. A protected operation can either be a subprogram or an entry. A protected entry specifies a Boolean expression (an entry barrier) that must be True before the body of the entry is executed. A protected unit may define a single protected object or a protected type permitting the creation of several similar objects.

Declarations and Statements 20

The body of a program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a sequence of statements, which defines the execution of the program unit. 21

The declarative part associates names with declared entities. For example, a name may denote a type, a constant, a variable, or an exception. A declarative part also introduces the names and parameters of other nested subprograms, packages, task units, protected units, and generic units to be used in the program unit. 22

The sequence of statements describes a sequence of actions that are to be performed. The statements are executed in succession (unless a transfer of control causes execution to continue from another place). 23

An assignment statement changes the value of a variable. A procedure call invokes execution of a procedure after associating any actual parameters provided at the call with the corresponding formal parameters. 24

Case statements and if statements allow the selection of an enclosed sequence of statements based on the value of an expression or on the value of a condition. 25

The loop statement provides the basic iterative mechanism in the language. A loop statement specifies that a sequence of statements is to be executed repeatedly as directed by an iteration scheme, or until an exit statement is encountered. 26

A block statement comprises a sequence of statements preceded by the declaration of local entities used by the statements. 27

Certain statements are associated with concurrent execution. A delay statement delays the execution of a task for a specified duration or until a specified time. An entry call statement is written as a procedure call statement; it requests an operation on a task or on a protected object, blocking the caller until the operation can be performed. A called task may accept an entry call by executing a corresponding accept statement, which specifies the actions then to be performed as part of the rendezvous with the calling task. An entry call on a protected object is processed when the corresponding entry barrier evaluates to true, whereupon the body of the entry is executed. The requeue statement permits the provision of a service as a number of related activities with preference control. One form of the select statement allows a selective wait for one of several alternative rendezvous. Other forms of the select statement allow conditional or timed entry calls and the asynchronous transfer of control in response to some triggering event. 28

Execution of a program unit may encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation may exceed the maximum allowed value of a number, or an attempt may be made to access an array component by using an incorrect index value. To deal with such error situations, the statements of a program unit can be textually followed by exception handlers that specify the actions to be taken when the error situation arises. Exceptions can be raised explicitly by a raise statement. 29

Data Types 30

Every object in the language has a type, which characterizes a set of values and a set of applicable operations. The main classes of types are elementary types (comprising enumeration, numeric, and access types) and composite types (including array and record types). 31

{*AI95-00285-01*} {*AI95-00387-01*} An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types Boolean, Character, Wide_Character, and Wide_Wide_Character are predefined. 32/2

33 Numeric types provide a means of performing exact or approximate numerical computations. Exact computations use integer types, which denote sets of consecutive integers. Approximate computations use either fixed point types, with absolute bounds on the error, or floating point types, with relative bounds on the error. The numeric types Integer, Float, and Duration are predefined.

34/2 {*AI95-00285-01*} {*AI95-00387-01*} Composite types allow definitions of structured objects with related components. The composite types in the language include arrays and records. An array is an object with indexed components of the same type. A record is an object with named components of possibly different types. Task and protected types are also forms of composite types. The array types String, Wide_String, and Wide_Wide_String are predefined.

35 Record, task, and protected types may have special components called discriminants which parameterize the type. Variant record structures that depend on the values of discriminants can be defined within a record type.

36 Access types allow the construction of linked data structures. A value of an access type represents a reference to an object declared as aliased or to an object created by the evaluation of an allocator. Several variables of an access type may designate the same object, and components of one object may designate the same or other objects. Both the elements in such linked data structures and their relation to other elements can be altered during program execution. Access types also permit references to subprograms to be stored, passed as parameters, and ultimately dereferenced as part of an indirect call.

37 Private types permit restricted views of a type. A private type can be defined in a package so that only the logically necessary properties are made visible to the users of the type. The full structural details that are externally irrelevant are then only available within the package and any child units.

38 From any type a new type may be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations may be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives may be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension must be tagged, so that the specific type of an operand within a derivation class can be identified at run time. When an operation of a tagged type is applied to an operand whose specific type is not known until run time, implicit dispatching is performed based on the tag of the operand.

38.1/2 {*AI95-00387-01*} Interface types provide abstract models from which other interfaces and types may be composed and derived. This provides a reliable form of multiple inheritance. Interface types may also be implemented by task types and protected types thereby enabling concurrent programming and inheritance to be merged.

39 The concept of a type is further refined by the concept of a subtype, whereby a user can constrain the set of allowed values of a type. Subtypes can be used to define subranges of scalar types, arrays with a limited set of index values, and records and private types with particular discriminant values.

40 *Other Facilities*

41/2 {*AI95-00387-01*} Aspect clauses can be used to specify the mapping between types and features of an underlying machine. For example, the user can specify that objects of a given type must be represented with a given number of bits, or that the components of a record are to be represented using a given storage layout. Other features allow the controlled use of low level, nonportable, or implementation-dependent aspects, including the direct insertion of machine code.

42/2 {*AI95-00387-01*} The predefined environment of the language provides for input-output and other capabilities by means of standard library packages. Input-output is supported for values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided.

{*AI95-00387-01*} The predefined standard library packages provide facilities such as string manipulation, containers of various kinds (vectors, lists, maps, etc.), mathematical functions, random number generation, and access to the execution environment. 42.1/2

{*AI95-00387-01*} The specialized annexes define further predefined library packages and facilities with emphasis on areas such as real-time scheduling, interrupt handling, distributed systems, numerical computation, and high-integrity systems. 42.2/2

Finally, the language provides a powerful means of parameterization of program units, called generic program units. The generic parameters can be types and subprograms (as well as objects and packages) and so allow general algorithms and data structures to be defined that are applicable to all types of a given class. 43

Language Changes

{*AI95-00387-01*} This amended International Standard updates the edition of 1995 which replaced the first edition of 1987. In the 1995 edition, the following major language changes were incorporated: 44/2

- {*AI95-00387-01*} Support for standard 8-bit and 16-bit characters was added. See clauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4. 45/2
- {*AI95-00387-01*} The type model was extended to include facilities for object-oriented programming with dynamic polymorphism. See the discussions of classes, derived types, tagged types, record extensions, and private extensions in clauses 3.4, 3.9, and 7.3. Additional forms of generic formal parameters were allowed as described in clauses 12.5.1 and 12.7. 46/2
- {*AI95-00387-01*} Access types were extended to allow an access value to designate a subprogram or an object declared by an object declaration as opposed to just an object allocated on a heap. See clause 3.10. 47/2
- {*AI95-00387-01*} Efficient data-oriented synchronization was provided by the introduction of protected types. See clause 9.4. 48/2
- {*AI95-00387-01*} The library structure was extended to allow library units to be organized into a hierarchy of parent and child units. See clause 10.1. 49/2
- {*AI95-00387-01*} Additional support was added for interfacing to other languages. See Annex B. 50/2
- {*AI95-00387-01*} The Specialized Needs Annexes were added to provide specific support for certain application areas: 51/2
  - Annex C, “Systems Programming” 52
  - Annex D, “Real-Time Systems” 53
  - Annex E, “Distributed Systems” 54
  - Annex F, “Information Systems” 55
  - Annex G, “Numerics” 56
  - Annex H, “High Integrity Systems” 57

{*AI95-00387-01*} Amendment 1 modifies the 1995 International Standard by making changes and additions that improve the capability of the language and the reliability of programs written in the language. In particular the changes were designed to improve the portability of programs, interfacing to other languages, and both the object-oriented and real-time capabilities. 57.1/2

{*AI95-00387-01*} The following significant changes with respect to the 1995 edition are incorporated: 57.2/2

57.3/2
- Support for program text is extended to cover the entire ISO/IEC 10646:2003 repertoire. Execution support now includes the 32-bit character set. See clauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4.

57.4/2
- The object-oriented model has been improved by the addition of an interface facility which provides multiple inheritance and additional flexibility for type extensions. See clauses 3.4, 3.9, and 7.3. An alternative notation for calling operations more akin to that used in other languages has also been added. See clause 4.1.3.

57.5/2
- Access types have been further extended to unify properties such as the ability to access constants and to exclude null values. See clause 3.10. Anonymous access types are now permitted more freely and anonymous access-to-subprogram types are introduced. See clauses 3.3, 3.6, 3.10, and 8.5.1.

57.6/2
- The control of structure and visibility has been enhanced to permit mutually dependent references between units and finer control over access from the private part of a package. See clauses 3.10.1 and 10.1.2. In addition, limited types have been made more useful by the provision of aggregates, constants, and constructor functions. See clauses 4.3, 6.5, and 7.5.

57.7/2
- The predefined environment has been extended to include additional time and calendar operations, improved string handling, a comprehensive container library, file and directory management, and access to environment variables. See clauses 9.6.1, A.4, A.16, A.17, and A.18.

57.8/2
- Two of the Specialized Needs Annexes have been considerably enhanced:

57.9/2
  - The Real-Time Systems Annex now includes the Ravenscar profile for high-integrity systems, further dispatching policies such as Round Robin and Earliest Deadline First, support for timing events, and support for control of CPU time utilization. See clauses D.2, D.13, D.14, and D.15.

57.10/2
  - The Numerics Annex now includes support for real and complex vectors and matrices as previously defined in ISO/IEC 13813:1997 plus further basic operations for linear algebra. See clause G.3.

57.11/2
- The overall reliability of the language has been enhanced by a number of improvements. These include new syntax which detects accidental overloading, as well as pragmas for making assertions and giving better control over the suppression of checks. See clauses 6.1, 11.4.2, and 11.5.

Instructions for Comment Submission

{*instructions for comment submission*} {*comments, instructions for submission*} Informal comments on this International Standard may be sent via e-mail to **ada-comment@ada-auth.org**. If appropriate, the Project Editor will initiate the defect correction procedure. 58/1

Comments should use the following format: 59

!topic *Title summarizing comment* 60/2
!reference Ada 2005 RM*ss.ss(pp)*
!from *Author Name yy-mm-dd*
!keywords *keywords related to topic*
!discussion

text of discussion

where *ss.ss* is the section, clause or subclause number, *pp* is the paragraph number where applicable, and *yy-mm-dd* is the date the comment was sent. The date is optional, as is the **!keywords** line. 61

Please use a descriptive “Subject” in your e-mail message, and limit each message to a single comment. 62/1

When correcting typographical errors or making minor wording suggestions, please put the correction directly as the topic of the comment; use square brackets [ ] to indicate text to be omitted and curly braces { } to indicate text to be added, and provide enough context to make the nature of the suggestion self-evident or put additional information in the body of the comment, for example: 63

!topic [c]{C}haracter 64
!topic it[']s meaning is not defined

Formal requests for interpretations and for reporting defects in this International Standard may be made in accordance with the ISO/IEC JTC 1 Directives and the ISO/IEC JTC 1/SC 22 policy for interpretations. National Bodies may submit a Defect Report to ISO/IEC JTC 1/SC 22 for resolution under the JTC 1 procedures. A response will be provided and, if appropriate, a Technical Corrigendum will be issued in accordance with the procedures. 65

Acknowledgements for the Ada 95 edition of the Ada Reference Manual

66 This International Standard was prepared by the Ada 9X Mapping/Revision Team based at Intermetrics, Inc., which has included: W. Carlson, Program Manager; T. Taft, Technical Director; J. Barnes (consultant); B. Brosgol (consultant); R. Duff (Oak Tree Software); M. Edwards; C. Garrity; R. Hilliard; O. Pazy (consultant); D. Rosenfeld; L. Shafer; W. White; M. Woodger.

67 The following consultants to the Ada 9X Project contributed to the Specialized Needs Annexes: T. Baker (Real-Time/Systems Programming — SEI, FSU); K. Dritz (Numerics — Argonne National Laboratory); A. Gargaro (Distributed Systems — Computer Sciences); J. Goodenough (Real-Time/Systems Programming — SEI); J. McHugh (Secure Systems — consultant); B. Wichmann (Safety-Critical Systems — NPL: UK).

68 This work was regularly reviewed by the Ada 9X Distinguished Reviewers and the members of the Ada 9X Rapporteur Group (XRG): E. Ploedereder, Chairman of DRs and XRG (University of Stuttgart: Germany); B. Bardin (Hughes); J. Barnes (consultant: UK); B. Brett (DEC); B. Brosgol (consultant); R. Brukardt (RR Software); N. Cohen (IBM); R. Dewar (NYU); G. Dismukes (TeleSoft); A. Evans (consultant); A. Gargaro (Computer Sciences); M. Gerhardt (ESL); J. Goodenough (SEI); S. Heilbrunner (University of Salzburg: Austria); P. Hilfinger (UC/Berkeley); B. Källberg (CelsiusTech: Sweden); M. Kamrad II (Unisys); J. van Katwijk (Delft University of Technology: The Netherlands); V. Kaufman (Russia); P. Kruchten (Rational); R. Landwehr (CCI: Germany); C. Lester (Portsmouth Polytechnic: UK); L. Månsson (TELIA Research: Sweden); S. Michell (Multiprocessor Toolsmiths: Canada); M. Mills (US Air Force); D. Pogge (US Navy); K. Power (Boeing); O. Roubine (Verdix: France); A. Strohmeier (Swiss Fed Inst of Technology: Switzerland); W. Taylor (consultant: UK); J. Tokar (Tartan); E. Vasilescu (Grumman); J. Vladik (Prospeks s.r.o.: Czech Republic); S. Van Vlierberghe (OFFIS: Belgium).

69 Other valuable feedback influencing the revision process was provided by the Ada 9X Language Precision Team (Odyssey Research Associates), the Ada 9X User/Implementer Teams (AETECH, Tartan, TeleSoft), the Ada 9X Implementation Analysis Team (New York University) and the Ada community-at-large.

70 Special thanks go to R. Mathis, Convenor of ISO/IEC JTC 1/SC 22 Working Group 9.

71 The Ada 9X Project was sponsored by the Ada Joint Program Office. Christine M. Anderson at the Air Force Phillips Laboratory (Kirtland AFB, NM) was the project manager.

Acknowledgements for the Corrigendum version of the Ada Reference Manual

71.1/1 The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this revision possible.

71.2/1 Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording corrections was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, E. Ploedereder (Germany), who kept the process moving; J. Barnes (UK) and K. Ishihata (Japan), whose extremely detailed reviews kept the editor on his toes; G. Dismukes (USA), M. Kamrad (USA), P. Leroy (France), S. Michell (Canada), T. Taft (USA), J. Tokar (USA), and other members too numerous to mention.

71.3/1 Special thanks go to R. Duff (USA) for his explanations of the previous system of formatting of these documents during the tedious conversion to more modern formats. Special thanks also go to the convener of ISO/IEC JTC 1/SC 22/WG 9, J. Moore (USA), without whose help and support the corrigendum and this consolidated reference manual would not have been possible.

Acknowledgements for the Amendment version of the Ada Reference Manual

71.4/2 The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this revision possible.

71.5/2 Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording corrections was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, P. Leroy (France), who kept the process on schedule; J. Barnes (UK) whose careful reviews found many typographical errors; T. Taft (USA), who always seemed to have a suggestion when we were stuck, and who also was usually able to provide the valuable service of explaining why things were as they are; S. Baird (USA), who found many obscure problems with the proposals; and A. Burns (UK), who pushed many of the real-time proposals to completion. Other ARG members who contributed were: R. Dewar (USA), G. Dismukes (USA), R. Duff (USA), K. Ishihata (Japan), S. Michell (Canada), E. Ploedereder (Germany), J.P. Rosen (France), E. Schonberg (USA), J. Tokar (USA), and T. Vardanega (Italy).

71.6/2 Special thanks go to Ada-Europe and the Ada Resource Association, without whose help and support the Amendment and this consolidated reference manual would not have been possible. M. Heaney (USA) requires special thanks for his tireless work on the containers packages. Finally, special thanks go to the convener of ISO/IEC JTC 1/SC 22/WG 9, J. Moore (USA), who guided the document through the standardization process.

Changes

72 The International Standard is the same as this version of the Reference Manual, except:

- 73 This list of Changes is not included in the International Standard.
- 74 The “Acknowledgements” page is not included in the International Standard.
- 75 The text in the running headers and footers on each page is slightly different in the International Standard.
- 76 The title page(s) are different in the International Standard.
- 77 This document is formatted for 8.5-by-11-inch paper, whereas the International Standard is formatted for A4 paper (210-by-297mm); thus, the page breaks are in different places.
- 77.1/1 The “Foreword to this version of the Ada Reference Manual” clause is not included in the International Standard.
- 77.2/2 The “Using this version of the Ada Reference Manual” clause is not included in the International Standard.

Using this version of the Ada Reference Manual

77.3/2 This document has been revised with the corrections specified in Technical Corrigendum 1 (ISO/IEC 8652:1995/COR.1:2001) and Amendment 1 (ISO/IEC 8652/AMD.1:2007). In addition, additional annotations have been added and a variety of editorial errors have been corrected.

77.4/2 Changes to the original 8652:1995 can be identified by the version number following the paragraph number. Paragraphs with a version number of /1 were changed by Technical Corrigendum 1 or were editorial corrections at that time, while paragraphs with a version number of /2 were changed by Amendment 1 or were more recent editorial corrections. Paragraphs not so marked are unchanged by Amendment 1, Technical Corrigendum 1, or editorial corrections; that is, they are identical to the 1995 version. Paragraph numbers of unchanged paragraphs are the same as in the original International Standard. Inserted text is indicated by underlining, and deleted text is indicated by strikethroughs. Some versions also use color to indicate the version of the change.Where paragraphs are inserted, the paragraph numbers are of the form pp.nn, where pp is the number of the preceding paragraph, and nn is an insertion number. For instance, the first paragraph inserted after paragraph 8 is numbered 8.1, the second paragraph inserted is numbered 8.2, and so on. Deleted paragraphs are indicated by the text *This paragraph was deleted.* Deleted paragraphs include empty paragraphs that were numbered in the original Ada Reference Manual. Similar markings and numbering is used for changes to annotations.

INTERNATIONAL STANDARD **ISO/IEC 8652:2007(E), Ed. 3**

Information technology — Programming Languages — Ada

Section 1: General

Ada is a programming language designed to support the construction of long-lived, highly reliable software systems. The language includes facilities to define packages of related types, objects, and operations. The packages may be parameterized and the types may be extended to support the construction of libraries of reusable, adaptable software components. The operations may be implemented as subprograms using conventional sequential control structures, or as entries that include synchronization of concurrent threads of control as part of their invocation. The language treats modularity in the physical sense as well, with a facility to support separate compilation. 1

The language includes a complete facility for the support of real-time, concurrent programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, and random number generation. 2

Discussion: This Annotated Ada Reference Manual (AARM) contains the entire text of the Ada Reference Manual with Amendment 1 (the Ada 2005 RM), plus certain annotations. The annotations give a more in-depth analysis of the language. They describe the reason for each non-obvious rule, and point out interesting ramifications of the rules and interactions among the rules (interesting to language lawyers, that is). Differences between Ada 83, Ada 95, and Ada 2005 are listed. (The text you are reading now is an annotation.) 2.a/2

The AARM stresses detailed correctness and uniformity over readability and understandability. We're not trying to make the language “appear” simple here; on the contrary, we're trying to expose hidden complexities, so we can more easily detect language bugs. The Ada 2005 RM, on the other hand, is intended to be a more readable document for programmers. 2.b/2

The annotations in the AARM are as follows: 2.c

- Text that is logically redundant is shown [in square brackets, like this]. Technically, such text could be written as a Note in the Ada 2005 RM (and the Ada 95 RM before it), since it is really a theorem that can be proven from the non-redundant rules of the language. We use the square brackets instead when it seems to make the Ada 2005 RM more readable. 2.d/2
- The rules of the language (and some AARM-only text) are categorized, and placed under certain *sub-headings* that indicate the category. For example, the distinction between Name Resolution Rules and Legality Rules is particularly important, as explained in 8.6. 2.e
- Text under the following sub-headings appears in both documents: 2.f
  - The unlabeled text at the beginning of each clause or subclause, 2.g
  - Syntax, 2.h
  - Name Resolution Rules, 2.i
  - Legality Rules, 2.j

2.k
  - Static Semantics,

2.l
  - Post-Compilation Rules,

2.m
  - Dynamic Semantics,

2.n
  - Bounded (Run-Time) Errors,

2.o
  - Erroneous Execution,

2.p
  - Implementation Requirements,

2.q
  - Documentation Requirements,

2.r
  - Metrics,

2.s
  - Implementation Permissions,

2.t
  - Implementation Advice,

2.u
  - NOTES,

2.v
  - Examples.

2.w/2
- Text under the following sub-headings does not appear in the Ada 2005 RM:

2.x
  - Language Design Principles,

2.y
  - Inconsistencies With Ada 83,

2.z
  - Incompatibilities With Ada 83,

2.aa
  - Extensions to Ada 83,

2.bb/2
  - Wording Changes from Ada 83,

2.bb.1/2
  - Inconsistencies With Ada 95,

2.bb.2/2
  - Incompatibilities With Ada 95,

2.bb.3/2
  - Extensions to Ada 95,

2.bb.4/2
  - Wording Changes from Ada 95.

2.cc
- The AARM also includes the following kinds of annotations. These do not necessarily annotate the immediately preceding rule, although they often do.

2.dd **Reason:** An explanation of why a certain rule is necessary, or why it is worded in a certain way.

2.ee **Ramification:** An obscure ramification of the rules that is of interest only to language lawyers. (If a ramification of the rules is of interest to programmers, then it appears under NOTES.)

2.ff **Proof:** An informal proof explaining how a given Note or [marked-as-redundant] piece of text follows from the other rules of the language.

2.gg **Implementation Note:** A hint about how to implement a feature, or a particular potential pitfall that an implementer needs to be aware of.

2.hh **Change:** Change annotations are not used in this version. Changes from previous versions have been removed. Changes in this version are marked with versioned paragraph numbers, as explained in the “Corrigendum Changes” clause of the “Introduction”.

2.ii **Discussion:** Other annotations not covered by the above.

2.jj **To be honest:** A rule that is considered logically necessary to the definition of the language, but which is so obscure or pedantic that only a language lawyer would care. These are the only annotations that could be considered part of the language definition.

2.kk **Glossary entry:** The text of a Glossary entry — this text will also appear in Annex N, “Glossary”.

2.ll/2 **Discussion:** In general, the Ada 2005 RM text appears in the normal font, whereas AARM-only text appears in a smaller font. Notes also appear in the smaller font, as recommended by ISO/IEC style guidelines. Ada examples are also usually printed in a smaller font.

2.mm If you have trouble finding things, be sure to use the index. {*italics, like this*} Each defined term appears there, and also in *italics, like this*. Syntactic categories defined in BNF are also indexed.

2.nn A definition marked “[distributed]” is the main definition for a term whose complete definition is given in pieces distributed throughout the document. The pieces are marked “[partial]” or with a phrase explaining what cases the partial definition applies to.

1.1 Scope

1 This International Standard specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs to a variety of data processing systems.

1.1.1 Extent

1 This International Standard specifies:

- 2 The form of a program written in Ada;
- 3 The effect of translating and executing such a program;
- 4 The manner in which program units may be combined to form Ada programs;
- 5 The language-defined library units that a conforming implementation is required to supply;
- 6 The permissible variations within the standard, and the manner in which they are to be documented;
- 7 Those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program containing such violations;
- 8 Those violations of the standard that a conforming implementation is not required to detect.

9 This International Standard does not specify:

- 10 The means whereby a program written in Ada is transformed into object code executable by a processor;
- 11 The means whereby translation or execution of programs is invoked and the executing units are controlled;
- 12 The size or speed of the object code, or the relative execution speed of different language constructs;
- 13 The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages;
- 14 The effect of unspecified execution.
- 15 The size of a program or program unit that will exceed the capacity of a particular conforming implementation.

1.1.2 Structure

1 This International Standard contains thirteen sections, fourteen annexes, and an index.

2 {*core language*} The *core* of the Ada language consists of:

- 3 Sections 1 through 13
- 4 Annex A, “Predefined Language Environment”
- 5 Annex B, “Interface to Other Languages”
- 6 Annex J, “Obsolescent Features”

7 {*Specialized Needs Annexes*} {*Annex (Specialized Needs)*} {*application areas*} The following *Specialized Needs Annexes* define features that are needed by certain application areas:

- 8 Annex C, “Systems Programming”

9 • Annex D, “Real-Time Systems”

10 • Annex E, “Distributed Systems”

11 • Annex F, “Information Systems”

12 • Annex G, “Numerics”

13 • Annex H, “High Integrity Systems”

14 {*normative*} {*Annex (normative)*} The core language and the Specialized Needs Annexes are normative, except that the material in each of the items listed below is informative:

15 • Text under a NOTES or Examples heading.

16 • Each clause or subclause whose title starts with the word “Example” or “Examples”.

17 All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Specialized Needs Annexes.

18 {*informative*} {*non-normative: See informative*} {*Annex (informative)*} The following Annexes are informative:

19 • Annex K, “Language-Defined Attributes”

20 • Annex L, “Language-Defined Pragmas”

21 • M.2, “Implementation-Defined Characteristics”

22 • Annex N, “Glossary”

23 • Annex P, “Syntax Summary”

23.a **Discussion:** The idea of the Specialized Needs Annexes is that implementations can choose to target certain application areas. For example, an implementation specifically targeted to embedded machines might support the application-specific features for Real-time Systems, but not the application-specific features for Information Systems.

23.b The Specialized Needs Annexes extend the core language only in ways that users, implementations, and standards bodies are allowed to extend the language; for example, via additional library units, attributes, representation items (see 13.1), pragmas, and constraints on semantic details that are left unspecified by the core language. Many implementations already provide much of the functionality defined by Specialized Needs Annexes; our goal is to increase uniformity among implementations by defining standard ways of providing the functionality.

23.c/2 {*AI95-00114-01*} We recommend that the certification procedures allow implementations to certify the core language, plus any set of the Specialized Needs Annexes. We recommend that implementations *not* be allowed to certify a portion of one of the Specialized Needs Annexes, although implementations can, of course, provide uncertified support for such portions. We have designed the Specialized Needs Annexes assuming that this recommendation is followed. Thus, our decisions about what to include and what not to include in those annexes are based on the assumption that each annex is certified in an “all-or-nothing” manner.

23.d An implementation may, of course, support extensions that are different from (but possibly related to) those defined by one of the Specialized Needs Annexes. We recommend that, where appropriate, implementations do this by adding library units that are children of existing language-defined library packages.

23.e An implementation should not provide extensions that conflict with those defined in the Specialized Needs Annexes, in the following sense: Suppose an implementation supports a certain error-free program that uses only functionality defined in the core and in the Specialized Needs Annexes. The implementation should ensure that that program will still be error free in some possible full implementation of all of the Specialized Needs Annexes, and that the semantics of the program will not change. For example, an implementation should not provide a package with the same name as one defined in one of the Specialized Needs Annexes, but that behaves differently, *even if that implementation does not claim conformance to that Annex*.

23.f Note that the Specialized Needs Annexes do not conflict with each other; it is the intent that a single implementation can conform to all of them.

24 Each section is divided into clauses and subclauses that have a common structure. Each section, clause, and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

Language Design Principles

24.a These are not rules of the language, but guiding principles or goals used in defining the rules of the language. In some cases, the goal is only partially met; such cases are explained.

24.b/2 This is not part of the definition of the language, and does not appear in the Ada 2005 RM.

Syntax

25 {*syntax (under Syntax heading)*} {*grammar (under Syntax heading)*} {*context free grammar (under Syntax heading)*} {*BNF (Backus-Naur Form) (under Syntax heading)*} {*Backus-Naur Form (BNF) (under Syntax heading)*} Syntax rules (indented).

Name Resolution Rules

26 {*name resolution rules*} {*overloading rules*} {*resolution rules*} Compile-time rules that are used in name resolution, including overload resolution.

26.a **Discussion:** These rules are observed at compile time. (We say "observed" rather than "checked," because these rules are not individually checked. They are really just part of the Legality Rules in Section 8 that require exactly one interpretation of each constituent of a complete context.) The only rules used in overload resolution are the Syntax Rules and the Name Resolution Rules.

26.b When dealing with non-overloadable declarations it sometimes makes no semantic difference whether a given rule is a Name Resolution Rule or a Legality Rule, and it is sometimes difficult to decide which it should be. We generally make a given rule a Name Resolution Rule only if it has to be. For example, "The name, if any, in a raise_statement shall be the name of an exception." is under "Legality Rules."

Legality Rules

27 {*legality rules*} {*compile-time error*} {*error (compile-time)*} Rules that are enforced at compile time. {*legal (construct)*} {*illegal (construct)*} A construct is *legal* if it obeys all of the Legality Rules.

27.a **Discussion:** These rules are not used in overload resolution.

27.b Note that run-time errors are always attached to exceptions; for example, it is not "illegal" to divide by zero, it just raises an exception.

Static Semantics

28 {*static semantics*} {*compile-time semantics*} A definition of the compile-time effect of each construct.

28.a **Discussion:** The most important compile-time effects represent the effects on the symbol table associated with declarations (implicit or explicit). In addition, we use this heading as a bit of a grab bag for equivalences, package specifications, etc. For example, this is where we put statements like so-and-so is equivalent to such-and-such. (We ought to try to really mean it when we say such things!) Similarly, statements about magically-generated implicit declarations go here. These rules are generally written as statements of fact about the semantics, rather than as a you-shall-do-such-and-such sort of thing.

Post-Compilation Rules

29 {*post-compilation error*} {*post-compilation rules*} {*link-time error: See post-compilation error*} {*error (link-time)*} Rules that are enforced before running a partition. {*legal (partition)*} {*illegal (partition)*} A partition is legal if its compilation units are legal and it obeys all of the Post-Compilation Rules.

29.a **Discussion:** It is not specified exactly when these rules are checked, so long as they are checked for any given partition before that partition starts running. An implementation may choose to check some such rules at compile time, and reject compilation_units accordingly. Alternatively, an implementation may check such rules when the partition is created (usually known as "link time"), or when the partition is mapped to a particular piece of hardware (but before the partition starts running).

Dynamic Semantics

30 {*dynamic semantics*} {*run-time semantics*} {*run-time error*} {*error (run-time)*} A definition of the run-time effect of each construct.

30.a **Discussion:** This heading describes what happens at run time. Run-time checks, which raise exceptions upon failure, are described here. Each item that involves a run-time check is marked with the name of the check — these are the same check names that are used in a pragma Suppress. Principle: Every check should have a name, usable in a pragma Suppress.

Bounded (Run-Time) Errors

31 {*bounded error*} Situations that result in bounded (run-time) errors (see 1.1.5).

31.a **Discussion:** The "bounds" of each such error are described here — that is, we characterize the set of all possible behaviors that can result from a bounded error occurring at run time.

Erroneous Execution

32 {*erroneous execution*} Situations that result in erroneous execution (see 1.1.5).

Implementation Requirements

33 {*implementation requirements*} Additional requirements for conforming implementations.

33.a **Discussion:** ...as opposed to rules imposed on the programmer. An example might be, "The smallest representable duration, Duration'Small, shall not be greater than twenty milliseconds."

33.b It's really just an issue of how the rule is worded. We could write the same rule as "The smallest representable duration is an implementation-defined value less than or equal to 20 milliseconds" and then it would be under "Static Semantics."

Documentation Requirements

34 {*documentation requirements*} Documentation requirements for conforming implementations.

34.a **Discussion:** These requirements are beyond those that are implicitly specified by the phrase "implementation defined". The latter require documentation as well, but we don't repeat these cases under this heading. Usually this heading is used for when the description of the documentation requirement is longer and does not correspond directly to one, narrow normative sentence.

Metrics

35 {*metrics*} Metrics that are specified for the time/space properties of the execution of certain language constructs.

Implementation Permissions

36 {*implementation permissions*} Additional permissions given to the implementer.

36.a **Discussion:** For example, "The implementation is allowed to impose further restrictions on the record aggregates allowed in code statements." When there are restrictions on the permission, those restrictions are given here also. For example, "An implementation is allowed to restrict the kinds of subprograms that are allowed to be main subprograms. However, it shall support at least parameterless procedures." — we don't split this up between here and "Implementation Requirements."

Implementation Advice

37 {*implementation advice*} {*advice*} Optional advice given to the implementer. The word "should" is used to indicate that the advice is a recommendation, not a requirement. It is implementation defined whether or not a given recommendation is obeyed.

37.a/2 **Implementation defined:** Whether or not each recommendation given in Implementation Advice is followed — see M.3, "Implementation Advice" for a listing.

37.b/1 **Discussion:** The advice generally shows the intended implementation, but the implementer is free to ignore it. The implementer is the sole arbiter of whether or not the advice has been obeyed, if not, whether the reason is a good one, and whether the required documentation is sufficient. It would be wrong for the ACATS to enforce any of this advice.

37.c For example, "Whenever possible, the implementation should choose a value no greater than fifty microseconds for the smallest representable duration, Duration'Small."

We use this heading, for example, when the rule is so low level or implementation-oriented as to be untestable. We also use this heading when we wish to encourage implementations to behave in a certain way in most cases, but we do not wish to burden implementations by requiring the behavior. 37.d

NOTES

1 {*notes*} Notes emphasize consequences of the rules described in the (sub)clause or elsewhere. This material is informative. 38

Examples

Examples illustrate the possible forms of the constructs described. This material is informative. 39

Discussion: 39.a

The next three headings list all language changes between Ada 83 and Ada 95. Language changes are any change that changes the set of text strings that are legal Ada programs, or changes the meaning of any legal program. Wording changes, such as changes in terminology, are not language changes. Each language change falls into one of the following three categories:

Inconsistencies With Ada 83

{*inconsistencies with Ada 83*} This heading lists all of the upward inconsistencies between Ada 83 and Ada 95. Upward inconsistencies are situations in which a legal Ada 83 program is a legal Ada 95 program with different semantics. This type of upward incompatibility is the worst type for users, so we only tolerate it in rare situations. 39.b

(Note that the semantics of a program is not the same thing as the behavior of the program. Because of Ada's indeterminacy, the “semantics” of a given feature describes a *set* of behaviors that can be exhibited by that feature. The set can contain more than one allowed behavior. Thus, when we ask whether the semantics changes, we are asking whether the set of behaviors changes.) 39.c

This is not part of the definition of the language, and does not appear in the Ada 95 or Ada 2005 RM. 39.d/2

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} This heading lists all of the upward incompatibilities between Ada 83 and Ada 95, except for the ones listed under “Inconsistencies With Ada 83” above. These are the situations in which a legal Ada 83 program is illegal in Ada 95. We do not generally consider a change that turns erroneous execution into an exception, or into an illegality, to be upwardly incompatible. 39.e

This is not part of the definition of the language, and does not appear in the Ada 95 or Ada 2005 RM. 39.f/2

Extensions to Ada 83

{*extensions to Ada 83*} This heading is used to list all upward compatible language changes; that is, language extensions. These are the situations in which a legal Ada 95 program is not a legal Ada 83 program. The vast majority of language changes fall into this category. 39.g

This is not part of the definition of the language, and does not appear in the Ada 95 or Ada 2005 RM. 39.h/2

39.i

As explained above, the next heading does not represent any language change:

Wording Changes from Ada 83

{*wording changes from Ada 83*} This heading lists some of the non-semantic changes between the Ada 83 RM and the the Ada 95 RM. It is incomplete; we have not attempted to list all wording changes, but only the “interesting” ones. 39.j/2

This is not part of the definition of the language, and does not appear in the Ada 95 or Ada 2005 RM. 39.k/2

Discussion: 39.l/2

The next three headings list all language changes between Ada 95 and Ada 2005 (the language defined by the Ada 95 standard plus Technical Corrigendum 1 plus Amendment 1). Each language change falls into one of the following three categories:

Inconsistencies With Ada 95

{*inconsistencies with Ada 95*} This heading lists all of the upward inconsistencies between Ada 95 and Ada 2005. Upward inconsistencies are situations in which a legal Ada 95 program is a legal Ada 2005 program with different semantics. 39.m/2

39.n/2 Inconsistencies marked with **Corrigendum:**{*Corrigendum*} are corrections to the original Ada 95 definition introduced by Corrigendum 1. Inconsistencies marked with **Amendment Correction:**{*Amendment Correction*} are corrections to the original Ada 95 definition added by Amendment 1. Formally, these are inconsistencies caused by Ada Issues classified as Binding Interpretations; implementations of Ada 95 are supposed to follow these corrections, not the original flawed language definition. Thus, these strictly speaking are not inconsistencies between Ada 95 and Ada 2005. Practically, however, they very well may be, as early Ada 95 implementations may not follow the recommendation. Inconsistencies so marked are not portable between Ada 95 implementations, while usually Ada 2005 will have more clearly defined behavior. Therefore, we document these for completeness.

39.o/2 This is not part of the definition of the language, and does not appear in the Ada 2005 RM.

Incompatibilities With Ada 95

39.p/2 {*incompatibilities with Ada 95*} This heading lists all of the upward incompatibilities between Ada 95 and Ada 2005, except for the ones listed under “Inconsistencies With Ada 95” above. These are the situations in which a legal Ada 95 program is illegal in Ada 2005.

39.q/2 As with inconsistencies, incompatibilities marked with **Corrigendum:** are corrections to the original Ada 95 definition introduced by Corrigendum 1. Incompatibilities marked with **Amendment Correction:** are corrections to the original Ada 95 definition added by Amendment 1. Formally, these are incompatibilities caused by Ada Issues classified as Binding Interpretations; implementations of Ada 95 are supposed to follow these corrections, not the original flawed language definition. Thus, these strictly speaking are not incompatibilities between Ada 95 and Ada 2005. Practically, however, they very well may be, as early Ada 95 implementations may not follow the recommendation. Therefore, some Ada 95 implementations may be able to compile the examples, while others may not. In constrast, Ada 2005 compilers will have consistent behavior. Therefore, we document these for completeness.

39.r/2 This is not part of the definition of the language, and does not appear in the Ada 2005 RM.

Extensions to Ada 95

39.s/2 {*extensions to Ada 95*} This heading is used to list all upward compatible language changes; that is, language extensions. These are the situations in which a legal Ada 2005 program is not a legal Ada 95 program. The vast majority of language changes fall into this category.

39.t/2 As with incompatibilities, extensions marked with **Corrigendum:** are corrections to the original Ada 95 definition introduced by Corrigendum 1. Extensions marked with **Amendment Correction:** are corrections to the original Ada 95 definition added by Amendment 1. Formally, these are extensions allowed by Ada Issues classified as Binding Interpretations. As corrections, implementations of Ada 95 are allowed to implement these extensions. Thus, these strictly speaking are not extensions of Ada 95; they're part of Ada 95. Practically, however, they very well may be extensions, as early Ada 95 implementations may not implement the extension. Therefore, some Ada 95 implementations may be able to compile the examples, while others may not. In constrast, Ada 2005 compilers will always support the extensions. Therefore, we document these for completeness.

39.u/2 This is not part of the definition of the language, and does not appear in the Ada 2005 RM.

39.v/2

As explained above, the next heading does not represent any language change:

Wording Changes from Ada 95

39.w/2 {*wording changes from Ada 95*} This heading lists some of the non-semantic changes between the Ada 95 RM and the Ada 2005 RM. This heading lists only “interesting” changes (for instance, editorial corrections are not listed). Changes which come from Technical Corrigendum 1 are marked **Corrigendum**; unmarked changes come from Amendment 1.

39.x/2 This is not part of the definition of the language, and does not appear in the Ada 2005 RM.

1.1.3 Conformity of an Implementation with the Standard

Implementation Requirements

1 {*conformance (of an implementation with the Standard)*} A conforming implementation shall:

1.a **Discussion:** {*implementation*} The *implementation* is the software and hardware that implements the language. This includes compiler, linker, operating system, hardware, etc.

1.b We first define what it means to “conform” in general — basically, the implementation has to properly implement the normative rules given throughout the standard. Then we define what it means to conform to a Specialized Needs Annex — the implementation must support the core features plus the features of that Annex. Finally, we define what it means to “conform to the Standard” — this requires support for the core language, and allows partial (but not conflicting) support for the Specialized Needs Annexes.

- Translate and correctly execute legal programs written in Ada, provided that they are not so large as to exceed the capacity of the implementation; 2
- Identify all programs or program units that are so large as to exceed the capacity of the implementation (or raise an appropriate exception at run time); 3

Implementation defined: Capacity limitations of the implementation. 3.a

- Identify all programs or program units that contain errors whose detection is required by this International Standard; 4

Discussion: Note that we no longer use the term “rejection” of programs or program units. We require that programs or program units with errors or that exceed some capacity limit be “identified”. The way in which errors or capacity problems are reported is not specified. 4.a

An implementation is allowed to use standard error-recovery techniques. We do not disallow such techniques from being used across compilation_unit or compilation boundaries. 4.b

See also the Implementation Requirements of 10.2, which disallow the execution of illegal partitions. 4.c

- Supply all language-defined library units required by this International Standard; 5

Implementation Note: An implementation cannot add to or modify the visible part of a language-defined library unit, except where such permission is explicitly granted, unless such modifications are semantically neutral with respect to the client compilation units of the library unit. An implementation defines the contents of the private part and body of language-defined library units. 5.a

An implementation can add with_clauses and use_clauses, since these modifications are semantically neutral to clients. (The implementation might need with_clauses in order to implement the private part, for example.) Similarly, an implementation can add a private part even in cases where a private part is not shown in the standard. Explicit declarations can be provided implicitly or by renaming, provided the changes are semantically neutral. 5.b

{*italics (implementation-defined)*} Wherever in the standard the text of a language-defined library unit contains an italicized phrase starting with “*implementation-defined*”, the implementation's version will replace that phrase with some implementation-defined text that is syntactically legal at that place, and follows any other applicable rules. 5.c

Note that modifications are permitted, even if there are other tools in the environment that can detect the changes (such as a program library browser), so long as the modifications make no difference with respect to the static or dynamic semantics of the resulting programs, as defined by the standard. 5.d

- Contain no variations except those explicitly permitted by this International Standard, or those that are impossible or impractical to avoid given the implementation's execution environment; 6

Implementation defined: Variations from the standard that are impractical to avoid given the implementation's execution environment. 6.a

Reason: The “impossible or impractical” wording comes from AI-325. It takes some judgement and common sense to interpret this. Restricting compilation units to less than 4 lines is probably unreasonable, whereas restricting them to less than 4 billion lines is probably reasonable (at least given today's technology). We do not know exactly where to draw the line, so we have to make the rule vague. 6.b

- Specify all such variations in the manner prescribed by this International Standard. 7

{*external effect (of the execution of an Ada program)*} {*effect (external)*} The *external effect* of the execution of an Ada program is defined in terms of its interactions with its external environment. {*external interaction*} The following are defined as *external interactions*: 8

- Any interaction with an external file (see A.7); 9
- The execution of certain code_statements (see 13.8); which code_statements cause external interactions is implementation defined. 10

Implementation defined: Which code_statements cause external interactions. 10.a

- Any call on an imported subprogram (see Annex B), including any parameters passed to it; 11
- Any result returned or exception propagated from a main subprogram (see 10.2) or an exported subprogram (see Annex B) to an external caller; 12

12.a **Discussion:** By "result returned" we mean to include function results and values returned in [**in**] **out** parameters.

12.a.1/1 {*8652/0094*} {*AI95-00119-01*} The lack of a result from a program that does not terminate is also included here.

13 • [Any read or update of an atomic or volatile object (see C.6);]

14 • The values of imported and exported objects (see Annex B) at the time of any other interaction with the external environment.

14.a **To be honest:** Also other uses of imported and exported entities, as defined by the implementation, if the implementation supports such pragmas.

15 A conforming implementation of this International Standard shall produce for the execution of a given Ada program a set of interactions with the external environment whose order and timing are consistent with the definitions and requirements of this International Standard for the semantics of the given program.

15.a **Ramification:** There is no need to produce any of the "internal effects" defined for the semantics of the program — all of these can be optimized away — so long as an appropriate sequence of external interactions is produced.

15.b **Discussion:** See also 11.6 which specifies various liberties associated with optimizations in the presence of language-defined checks, that could change the external effects that might be produced. These alternative external effects are still consistent with the standard, since 11.6 is part of the standard.

15.c Note also that we only require "*an appropriate* sequence of external interactions" rather than "*the same* sequence..." An optimizer may cause a different sequence of external interactions to be produced than would be produced without the optimizer, so long as the new sequence still satisfies the requirements of the standard. For example, optimization might affect the relative rate of progress of two concurrent tasks, thereby altering the order in which two external interactions occur.

15.d/2 Note that the Ada 83 RM explicitly mentions the case of an "exact effect" of a program, but since so few programs have their effects defined that exactly, we don't even mention this "special" case. In particular, almost any program that uses floating point or tasking has to have some level of inexactness in the specification of its effects. And if one includes aspects of the timing of the external interactions in the external effect of the program (as is appropriate for a real-time language), no "exact effect" can be specified. For example, if two external interactions initiated by a single task are separated by a "**delay** 1.0;" then the language rules imply that the two external interactions have to be separated in time by at least one second, as defined by the clock associated with the delay_relative_statement. This in turn implies that the time at which an external interaction occurs is part of the characterization of the external interaction, at least in some cases, again making the specification of the required "exact effect" impractical.

16 An implementation that conforms to this Standard shall support each capability required by the core language as specified. In addition, an implementation that conforms to this Standard may conform to one or more Specialized Needs Annexes (or to none). Conformance to a Specialized Needs Annex means that each capability required by the Annex is provided as specified.

16.a **Discussion:** The last sentence defines what it means to say that an implementation conforms to a Specialized Needs Annex, namely, only by supporting all capabilities required by the Annex.

17 An implementation conforming to this International Standard may provide additional attributes, library units, and pragmas. However, it shall not provide any attribute, library unit, or pragma having the same name as an attribute, library unit, or pragma (respectively) specified in a Specialized Needs Annex unless the provided construct is either as specified in the Specialized Needs Annex or is more limited in capability than that required by the Annex. A program that attempts to use an unsupported capability of an Annex shall either be identified by the implementation before run time or shall raise an exception at run time.

17.a **Discussion:** The last sentence of the preceding paragraph defines what an implementation is allowed to do when it does not "conform" to a Specialized Needs Annex. In particular, the sentence forbids implementations from providing a construct with the same name as a corresponding construct in a Specialized Needs Annex but with a different syntax (e.g., an extended syntax) or quite different semantics. The phrase concerning "more limited in capability" is intended to give permission to provide a partial implementation, such as not implementing a subprogram in a package or having a restriction not permitted by an implementation that conforms to the Annex. For example, a partial implementation of the package Ada.Decimal might have Decimal.Max_Decimal_Digits as 15 (rather than the required 18). This allows a partial implementation to grow to a fully conforming implementation.

17.b A restricted implementation might be restricted by not providing some subprograms specified in one of the packages defined by an Annex. In this case, a program that tries to use the missing subprogram will usually fail to compile. Alternatively, the implementation might declare the subprogram as abstract, so it cannot be called. {*Program_Error (raised by failure of run-time check)*} Alternatively, a subprogram body might be implemented just to raise Program_Error. The advantage of this approach is that a program to be run under a fully conforming Annex implementation can be checked syntactically and semantically under an implementation that only partially supports the Annex. Finally, an implementation might provide a package declaration without the corresponding body, so that programs can be compiled, but partitions cannot be built and executed.

17.c To ensure against wrong answers being delivered by a partial implementation, implementers are required to raise an exception when a program attempts to use an unsupported capability and this can be detected only at run time. For example, a partial implementation of Ada.Decimal might require the length of the Currency string to be 1, and hence, an exception would be raised if a subprogram were called in the package Edited_Output with a length greater than 1.

Documentation Requirements

18 {*implementation defined*} {*unspecified*} {*specified (not!)*} {*implementation-dependent: See unspecified*} {*documentation (required of an implementation)*} Certain aspects of the semantics are defined to be either *implementation defined* or *unspecified*. In such cases, the set of possible effects is specified, and the implementation may choose any effect in the set. Implementations shall document their behavior in implementation-defined situations, but documentation is not required for unspecified situations. The implementation-defined characteristics are summarized in M.2.

18.a **Discussion:** We used to use the term “implementation dependent” instead of “unspecified”. However, that sounded too much like “implementation defined”. Furthermore, the term “unspecified” is used in the ANSI C and POSIX standards for this purpose, so that is another advantage. We also use “not specified” and “not specified by the language” as synonyms for “unspecified.” The documentation requirement is the only difference between implementation defined and unspecified.

18.b Note that the “set of possible effects” can be “all imaginable effects”, as is the case with erroneous execution.

19 The implementation may choose to document implementation-defined behavior either by documenting what happens in general, or by providing some mechanism for the user to determine what happens in a particular case.

19.a **Discussion:** For example, if the standard says that library unit elaboration order is implementation defined, the implementation might describe (in its user's manual) the algorithm it uses to determine the elaboration order. On the other hand, the implementation might provide a command that produces a description of the elaboration order for a partition upon request from the user. It is also acceptable to provide cross references to existing documentation (for example, a hardware manual), where appropriate.

19.b Note that dependence of a program on implementation-defined or unspecified functionality is not defined to be an error; it might cause the program to be less portable, however.

19.c/2 **Documentation Requirement:** The behavior of implementations in implementation-defined situations shall be documented — see M.2, “Implementation-Defined Characteristics” for a listing.

Implementation Advice

20 {*Program_Error (raised by failure of run-time check)*} If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise Program_Error if feasible.

20.a.1/2 **Implementation Advice:** Program_Error should be raised when an unsupported Specialized Needs Annex feature is used at run time.

20.a **Reason:** The reason we don't *require* Program_Error is that there are situations where other exceptions might make sense. For example, if the Real Time Systems Annex requires that the range of System.Priority include at least 30 values, an implementation could conform to the Standard (but not to the Annex) if it supported only 12 values. Since the rules of the language require Constraint_Error to be raised for out-of-range values, we cannot require Program_Error to be raised instead.

21 If an implementation wishes to provide implementation-defined extensions to the functionality of a language-defined library unit, it should normally do so by adding children to the library unit.

21.a.1/2 **Implementation Advice:** Implementation-defined extensions to the functionality of a language-defined library unit should be provided by adding children to the library unit.

21.a **Implementation Note:** If an implementation has support code ("run-time system code") that is needed for the execution of user-defined code, it can put that support code in child packages of System. Otherwise, it has to use some trick to avoid polluting the user's namespace. It is important that such tricks not be available to user-defined code (not in the standard mode, at least) — that would defeat the purpose.

NOTES

22 2 The above requirements imply that an implementation conforming to this Standard may support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities.

22.a **Discussion:** A conforming implementation can partially support a Specialized Needs Annex. Such an implementation does not conform to the Annex, but it does conform to the Standard.

1.1.4 Method of Description and Syntax Notation

1 The form of an Ada program is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules.

2 The meaning of Ada programs is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs.

3 {*syntax (notation)*} {*grammar (notation)*} {*context free grammar (notation)*} {*BNF (Backus-Naur Form) (notation)*} {*Backus-Naur Form (BNF) (notation)*} The context-free syntax of the language is described using a simple variant of Backus-Naur Form. In particular:

4 • Lower case words in a sans-serif font, some containing embedded underlines, are used to denote syntactic categories, for example:

5 case_statement

6 • Boldface words are used to denote reserved words, for example:

7 **array**

8 • Square brackets enclose optional items. Thus the two following rules are equivalent.

9/2 {*AI95-00433-01*} simple_return_statement ::= **return** [expression];
simple_return_statement ::= **return**; | **return** expression;

10 • Curly brackets enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent.

11 term ::= factor {multiplying_operator factor}
term ::= factor | term multiplying_operator factor

12 • A vertical line separates alternative items unless it occurs immediately after an opening curly bracket, in which case it stands for itself:

13 constraint ::= scalar_constraint | composite_constraint
discrete_choice_list ::= discrete_choice {| discrete_choice}

14 • {*italics (syntax rules)*} If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example *subtype_*name and *task_*name are both equivalent to name alone.

14.a **Discussion:** {*LR(1)*} {*ambiguous grammar*} {*grammar (resolution of ambiguity)*} {*grammar (ambiguous)*} The grammar given in this International Standard is not LR(1). In fact, it is ambiguous; the ambiguities are resolved by the overload resolution rules (see 8.6).

14.b We often use "if" to mean "if and only if" in definitions. For example, if we define "photogenic" by saying, "A type is photogenic if it has the following properties...," we mean that a type is photogenic if *and only if* it has those properties. It is usually clear from the context, and adding the "and only if" seems too cumbersome.

14.c When we say, for example, "a declarative_item of a declarative_part", we are talking about a declarative_item immediately within that declarative_part. When we say "a declarative_item in, or within, a declarative_part", we are talking about a declarative_item anywhere in the declarative_part, possibly deeply nested within other declarative_parts. (This notation doesn't work very well for names, since the name "of" something also has another meaning.)

14.d When we refer to the name of a language-defined entity (for example, Duration), we mean the language-defined entity even in programs where the declaration of the language-defined entity is hidden by another declaration. For example, when we say that the expected type for the expression of a delay_relative_statement is Duration, we mean the language-defined type Duration that is declared in Standard, not some type Duration the user might have declared.

14.1/2 {*AI95-00285-01*} The delimiters, compound delimiters, reserved words, and numeric_literals are exclusively made of the characters whose code position is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this International Standard (see 2.1) belong to the same range. [For example, the character E in the definition of exponent is the character whose name is "LATIN CAPITAL LETTER E", not "GREEK CAPITAL LETTER EPSILON".]

14.e/2 **Discussion:** This just means that programs can be written in plain ASCII characters; no characters outside of the 7-bit range are required.

14.2/2 {*AI95-00395-01*} When this International Standard mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using locale-independent full case folding, as defined by documents referenced in the note in section 1 of ISO/IEC 10646:2003.

14.f/2 **Discussion:** Unless otherwise specified for sequences of characters, case folding is applied to the sequence, not to individual characters. It sometimes can make a difference.

15 {*syntactic category*} A *syntactic category* is a nonterminal in the grammar defined in BNF under "Syntax." Names of syntactic categories are set in a different font, like_this.

16 {*Construct*} [Glossary Entry]A *construct* is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under "Syntax".

16.a **Ramification:** For example, an expression is a construct. A declaration is a construct, whereas the thing declared by a declaration is an "entity."

16.b **Discussion:** "Explicit" and "implicit" don't mean exactly what you might think they mean: The text of an instance of a generic is considered explicit, even though it does not appear explicitly (in the non-technical sense) in the program text, and even though its meaning is not defined entirely in terms of that text.

17 {*constituent (of a construct)*} A *constituent* of a construct is the construct itself, or any construct appearing within it.

18 {*arbitrary order*} Whenever the run-time semantics defines certain actions to happen in an *arbitrary order*, this means that the implementation shall arrange for these actions to occur in a way that is equivalent to some sequential order, following the rules that result from that sequential order. When evaluations are defined to happen in an arbitrary order, with conversion of the results to some subtypes, or with some run-time checks, the evaluations, conversions, and checks may be arbitrarily interspersed, so long as each expression is evaluated before converting or checking its value. {*type conversion (arbitrary order)* [partial]} {*conversion (arbitrary order)* [partial]} [Note that the effect of a program can depend on the order chosen by the implementation. This can happen, for example, if two actual parameters of a given call have side effects.]

18.a **Discussion:** Programs will be more portable if their external effect does not depend on the particular order chosen by an implementation.

18.b **Ramification:** Additional reordering permissions are given in 11.6, "Exceptions and Optimization".

18.c There is no requirement that the implementation always choose the same order in a given kind of situation. In fact, the implementation is allowed to choose a different order for two different executions of the same construct. However, we expect most implementations will behave in a relatively predictable manner in most situations.

18.d **Reason:** The "sequential order" wording is intended to allow the programmer to rely on "benign" side effects. For example, if F is a function that returns a unique integer by incrementing some global and returning the result, a call such as P(F, F) is OK if the programmer cares only that the two results of F are unique; the two calls of F cannot be executed in parallel, unless the compiler can prove that parallel execution is equivalent to some sequential order.

NOTES

19 3 The syntax rules describing structured constructs are presented in a form that corresponds to the recommended paragraphing. For example, an if_statement is defined as:

20
```
if_statement ::=
    if condition then
      sequence_of_statements
   {elsif condition then
      sequence_of_statements}
   [else
      sequence_of_statements]
    end if;
```

21 4 The line breaks and indentation in the syntax rules indicate the recommended line breaks and indentation in the corresponding constructs. The preferred places for other line breaks are after semicolons.

Wording Changes from Ada 95

21.a/2 {*AI95-00285-01*} We now explicitly say that the lexical elements of the language (with a few exceptions) are made up of characters in the lower half of the Latin-1 character set. This is needed to avoid confusion given the new capability to use most ISO 10646 characters in identifiers and strings.

21.b/2 {*AI95-00395-01*} We now explicitly define what the Standard means by upper case, as there are many possibilities for ISO 10646 characters.

21.c/2 {*AI95-00433-01*} The example for square brackets has been changed as there is no longer a return_statement syntax rule.

1.1.5 Classification of Errors

Implementation Requirements

1 The language definition classifies errors into several different categories:

2 • Errors that are required to be detected prior to run time by every Ada implementation;

3 These errors correspond to any violation of a rule given in this International Standard, other than those listed below. In particular, violation of any rule that uses the terms shall, allowed, permitted, legal, or illegal belongs to this category. Any program that contains such an error is not a legal Ada program; on the other hand, the fact that a program is legal does not mean, *per se*, that the program is free from other forms of error.

4 {*compile-time error*} {*error (compile-time)*} {*link-time error: See post-compilation error*} {*error (link-time)*} The rules are further classified as either compile time rules, or post compilation rules, depending on whether a violation has to be detected at the time a compilation unit is submitted to the compiler, or may be postponed until the time a compilation unit is incorporated into a partition of a program.

4.a **Ramification:** See, for example, 10.1.3, "Subunits of Compilation Units", for some errors that are detected only after compilation. Implementations are allowed, but not required, to detect post compilation rules at compile time when possible.

5 • Errors that are required to be detected at run time by the execution of an Ada program;

6 {*run-time error*} {*error (run-time)*} The corresponding error situations are associated with the names of the predefined exceptions. Every Ada compiler is required to generate code that raises the corresponding exception if such an error situation arises during program execution. [If such an error situation is certain to arise in every execution of a construct, then an implementation is allowed (although not required) to report this fact at compilation time.]

7 • Bounded errors;

8 The language rules define certain kinds of errors that need not be detected either prior to or during run time, but if not detected, the range of possible effects shall be bounded. {*bounded*

error} The errors of this category are called *bounded errors*. {*Program_Error (raised by failure of run-time check)*} The possible effects of a given bounded error are specified for each such error, but in any case one possible effect of a bounded error is the raising of the exception Program_Error.

- Erroneous execution. 9

{*erroneous execution*} In addition to bounded errors, the language rules define certain kinds of errors as leading to *erroneous execution*. Like bounded errors, the implementation need not detect such errors either prior to or during run time. Unlike bounded errors, there is no language-specified bound on the possible effect of erroneous execution; the effect is in general not predictable. 10

Ramification: Executions are erroneous, not programs or parts of programs. Once something erroneous happens, the execution of the entire program is erroneous from that point on, and potentially before given possible reorderings permitted by 11.6 and elsewhere. We cannot limit it to just one partition, since partitions are not required to live in separate address spaces. (But implementations are encouraged to limit it as much as possible.) 10.a

Suppose a program contains a pair of things that will be executed "in an arbitrary order." It is possible that one order will result in something sensible, whereas the other order will result in erroneous execution. If the implementation happens to choose the first order, then the execution is not erroneous. This may seem odd, but it is not harmful. 10.b

Saying that something is erroneous is semantically equivalent to saying that the behavior is unspecified. However, "erroneous" has a slightly more disapproving flavor. 10.c

Implementation Permissions

[{*mode of operation (nonstandard)*} {*nonstandard mode*} An implementation may provide *nonstandard modes* of operation. Typically these modes would be selected by a pragma or by a command line switch when the compiler is invoked. When operating in a nonstandard mode, the implementation may reject compilation_units that do not conform to additional requirements associated with the mode, such as an excessive number of warnings or violation of coding style guidelines. Similarly, in a nonstandard mode, the implementation may apply special optimizations or alternative algorithms that are only meaningful for programs that satisfy certain criteria specified by the implementation. {*mode of operation (standard)*} {*standard mode*} In any case, an implementation shall support a *standard* mode that conforms to the requirements of this International Standard; in particular, in the standard mode, all legal compilation_units shall be accepted.] 11

Discussion: These permissions are designed to authorize explicitly the support for alternative modes. Of course, nothing we say can prevent them anyway, but this (redundant) paragraph is designed to indicate that such alternative modes are in some sense "approved" and even encouraged where they serve the specialized needs of a given user community, so long as the standard mode, designed to foster maximum portability, is always available. 11.a

Implementation Advice

{*Program_Error (raised by failure of run-time check)*} If an implementation detects a bounded error or erroneous execution, it should raise Program_Error. 12

Implementation Advice: If a bounded error or erroneous execution is detected, Program_Error should be raised. 12.a.1/2

Wording Changes from Ada 83

Some situations that are erroneous in Ada 83 are no longer errors at all. For example, depending on the parameter passing mechanism when unspecified is possibly non-portable, but not erroneous. 12.a

Other situations that are erroneous in Ada 83 are changed to be bounded errors. In particular, evaluating an uninitialized scalar variable is a bounded error. {*Program_Error (raised by failure of run-time check)*} The possible results are to raise Program_Error (as always), or to produce a machine-representable value (which might not be in the subtype of the variable). {*Constraint_Error (raised by failure of run-time check)*} Violating a Range_Check or Overflow_Check raises Constraint_Error, even if the value came from an uninitialized variable. This means that optimizers can no longer "assume" that all variables are initialized within their subtype's range. Violating a check that is suppressed remains erroneous. 12.b

12.c The “incorrect order dependences” category of errors is removed. All such situations are simply considered potential non-portabilities. This category was removed due to the difficulty of defining what it means for two executions to have a “different effect.” For example, if a function with a side-effect is called twice in a single expression, it is not in principle possible for the compiler to decide whether the correctness of the resulting program depends on the order of execution of the two function calls. A compile time warning might be appropriate, but raising of Program_Error at run time would not be.

1.2 Normative References

1 {*references*} {*bibliography*} The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

2 {*ISO/IEC 646:1991*} {*646:1991, ISO/IEC standard*} {*character set standard (7-bit)*} ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*.

3/2 {*AI95-00415-01*} {*ISO/IEC 1539-1:2004*} {*1539-1:2004, ISO/IEC standard*} {*Fortran standard*} ISO/IEC 1539-1:2004, *Information technology — Programming languages — Fortran — Part 1: Base language*.

4/2 {*AI95-00415-01*} {*ISO 1989:2002*} {*1989:2002, ISO standard*} {*COBOL standard*} ISO/IEC 1989:2002, *Information technology — Programming languages — COBOL*.

5 {*ISO/IEC 6429:1992*} {*6429:1992, ISO/IEC standard*} {*character set standard (control functions)*} ISO/IEC 6429:1992, *Information technology — Control functions for coded graphic character sets*.

5.1/2 {*AI95-00351-01*} {*ISO 8601:2004*} {*date and time formatting standard*} ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

6 {*ISO/IEC 8859-1:1987*} {*8859-1:1987, ISO/IEC standard*} {*character set standard (8-bit)*} ISO/IEC 8859-1:1987, *Information processing — 8-bit single-byte coded character sets — Part 1: Latin alphabet No. 1*.

7/2 {*AI95-00415-01*} {*ISO/IEC 9899:1999*} {*9899:1999, ISO/IEC standard*} {*C standard*} ISO/IEC 9899:1999, *Programming languages — C*, supplemented by Technical Corrigendum 1:2001 and Technical Corrigendum 2:2004.

7.a/2 **Discussion:** Unlike Fortran and COBOL, which added the *Information technology* prefix to the titles of their standard, C did not. This was confirmed in the list of standards titles on the ISO web site. No idea why ISO allowed that.

8/2 {*8652/0001*} {*AI95-00124-01*} {*AI95-00285-01*} {*ISO/IEC 10646:2003*} {*10646:2003, ISO/IEC standard*} {*character set standard (16 and 32-bit)*} ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

8.a.1/2 *This paragraph was deleted.*{*8652/0001*} {*AI95-00124-01*} {*AI95-00285-01*}

9/2 {*AI95-00376-01*} {*ISO/IEC 14882:2003*} {*14882:2003, ISO/IEC standard*} {*C++ standard*} ISO/IEC 14882:2003, *Programming languages — C++*.

9.a/2 **Discussion:** This title is also missing the *Information technology* part. That was confirmed in the list of standards titles on the ISO web site.

10/2 {*AI95-00285-01*} {*ISO/IEC TR 19769:2004*} {*19769:2004, ISO/IEC technical report*} ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types*.

10.a **Discussion:** {*POSIX*} POSIX, *Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, The Institute of Electrical and Electronics Engineers, 1990.

Wording Changes from Ada 95

{*AI95-00285-01*} {*AI95-00376-01*} {*AI95-00415-01*} Updated references to the most recent versions of these standards. Added C++ and time standards. Added C character set technical report. 10.b/2

1.3 Definitions

{*AI95-00415-01*} {*italics (terms introduced or defined)*} Terms are defined throughout this International Standard, indicated by *italic* type. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Mathematical terms not defined in this International Standard are to be interpreted according to the *CRC Concise Encyclopedia of Mathematics, Second Edition*. Other terms not defined in this International Standard are to be interpreted according to the *Webster's Third New International Dictionary of the English Language*. Informal descriptions of some terms are also given in Annex N, “Glossary”. 1/2

Discussion: The index contains an entry for every defined term. 1.a

{*AI95-00415-01*} The contents of the *CRC Concise Encyclopedia of Mathematics, Second Edition* can be accessed on http://www.mathworld.com. The ISBN number of the book is ISBN 1584883472. 1.a.1/2

Glossary entry: Each term defined in Annex N is marked like this. 1.b

Discussion: Here are some AARM-only definitions: {*Ada Rapporteur Group (ARG)*} {*ARG*} The Ada Rapporteur Group (ARG) interprets the Ada Reference Manual. {*Ada Issue (AI)*} {*AI*} An Ada Issue (AI) is a numbered ruling from the ARG. Ada Issues created for Ada 83 are denoted as "AI83", while Ada Issues created for Ada 95 are denoted as "AI95" in this document. {*Ada Commentary Integration Document (ACID)*} {*ACID*} The Ada Commentary Integration Document (ACID) is an edition of the Ada 83 RM in which clearly marked insertions and deletions indicate the effect of integrating the approved AIs. {*Uniformity Rapporteur Group (URG)*} {*URG*} The Uniformity Rapporteur Group (URG) issued recommendations intended to increase uniformity across Ada implementations. The functions of the URG have been assumed by the ARG. {*Uniformity Issue (UI)*} {*UI*} A Uniformity Issue (UI) was a numbered recommendation from the URG. A Defect Report and Response is an official query to WG9 about an error in the standard. Defect Reports are processed by the ARG, and are referenced here by their ISO numbers: 8652/nnnn. Most changes to the Ada 95 standard include reference(s) to the Defect Report(s) that prompted the change. {*ACVC (Ada Compiler Validation Capability)* [partial]} {*Ada Compiler Validation Capability (ACVC)* [partial]} {*ACATS (Ada Conformity Assessment Test Suite)* [partial]} {*Ada Conformity Assessment Test Suite (ACATS)* [partial]} The *Ada Conformity Assessment Test Suite (ACATS)* is a set of tests intended to check the conformity of Ada implementations to this standard. This set of tests was previously known as the Ada Compiler Validation Capability (ACVC). 1.c/2

Section 2: Lexical Elements

1 [The text of a program consists of the texts of one or more compilations. The text of a compilation is a sequence of lexical elements, each composed of characters; the rules of composition are given in this section. Pragmas, which provide certain information for the compiler, are also described in this section.]

2.1 Character Set

1/2 {*AI95-00285-01*} {*AI95-00395-01*} {*character set*} The character repertoire for the text of an Ada program consists of the entire coding space described by the ISO/IEC 10646:2003 Universal Multiple-Octet Coded Character Set. This coding space is organized in *planes*, each plane comprising 65536 characters.{*plane (character)*} {*character plane*}

1.a/2 *This paragraph was deleted.*{*AI95-00285-01*}

1.b/2 *This paragraph was deleted.*{*AI95-00285-01*}

1.c/2 **Discussion:** {*AI95-00285-01*} It is our intent to follow the terminology of ISO/IEC 10646:2003 where appropriate, and to remain compatible with the character classifications defined in A.3, "Character Handling".

Syntax

Paragraphs 2 and 3 were deleted.

3.1/2 {*AI95-00285-01*} {*AI95-00395-01*} A character is defined by this International Standard for each cell in the coding space described by ISO/IEC 10646:2003, regardless of whether or not ISO/IEC 10646:2003 allocates a character to that cell.

Static Semantics

4/2 {*AI95-00285-01*} {*AI95-00395-01*} The coded representation for characters is implementation defined [(it need not be a representation defined within ISO/IEC 10646:2003)]. A character whose relative code position in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program.

4.a **Implementation defined:** The coded representation for the text of an Ada program.

4.b/2 **Ramification:** {*AI95-00285-01*} Note that this rule doesn't really have much force, since the implementation can represent characters in the source in any way it sees fit. For example, an implementation could simply define that what seems to be an other_private_use character is actually a representation of the space character.

4.1/2 {*AI95-00285-01*} The semantics of an Ada program whose text is not in Normalization Form KC (as defined by section 24 of ISO/IEC 10646:2003) is implementation defined.

4.c/2 **Implementation defined:** The semantics of an Ada program whose text is not in Normalization Form KC.

5/2 {*AI95-00285-01*} The description of the language definition in this International Standard uses the character properties General Category, Simple Uppercase Mapping, Uppercase Mapping, and Special Case Condition of the documents referenced by the note in section 1 of ISO/IEC 10646:2003. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified. {*unspecified* [partial]}

6/2 {*AI95-00285-01*} Characters are categorized as follows:

6.a/2 **Discussion:** Our character classification considers that the cells not allocated in ISO/IEC 10646:2003 are graphic characters, except for those whose relative code position in their plane is 16#FFFE# or 16#FFFF#. This seems to provide the best compatibility with future versions of ISO/IEC 10646, as future characters can be already be used in Ada character and string literals.

7/2 *This paragraph was deleted.*{*AI95-00285-01*}

8/2 {*AI95-00285-01*} {*letter_uppercase*} letter_uppercase
Any character whose General Category is defined to be "Letter, Uppercase".

9/2 {*AI95-00285-01*} {*letter_lowercase*} letter_lowercase
Any character whose General Category is defined to be “Letter, Lowercase”.

9.a/1 *This paragraph was deleted.*{*8652/0001*} {*AI95-00124-01*}

9.1/2 {*AI95-00285-01*} {*letter_titlecase*} letter_titlecase
Any character whose General Category is defined to be “Letter, Titlecase”.

9.2/2 {*AI95-00285-01*} {*letter_modifier*} letter_modifier
Any character whose General Category is defined to be “Letter, Modifier”.

9.3/2 {*AI95-00285-01*} {*letter_other*} letter_other
Any character whose General Category is defined to be “Letter, Other”.

9.4/2 {*AI95-00285-01*} {*mark_non_spacing*} mark_non_spacing
Any character whose General Category is defined to be “Mark, Non-Spacing”.

9.5/2 {*AI95-00285-01*} {*mark_non_spacing*} mark_spacing_combining
Any character whose General Category is defined to be “Mark, Spacing Combining”.

10/2 {*AI95-00285-01*} {*number_decimal*} number_decimal
Any character whose General Category is defined to be “Number, Decimal”.

10.1/2 {*AI95-00285-01*} {*number_letter*} number_letter
Any character whose General Category is defined to be “Number, Letter”.

10.2/2 {*AI95-00285-01*} {*punctuation_connector*} punctuation_connector
Any character whose General Category is defined to be “Punctuation, Connector”.

10.3/2 {*AI95-00285-01*} {*other_format*} other_format
Any character whose General Category is defined to be “Other, Format”.

11/2 {*AI95-00285-01*} {*separator_space*} separator_space
Any character whose General Category is defined to be “Separator, Space”.

12/2 {*AI95-00285-01*} {*separator_line*} separator_line
Any character whose General Category is defined to be “Separator, Line”.

12.1/2 {*AI95-00285-01*} {*separator_paragraph*} separator_paragraph
Any character whose General Category is defined to be “Separator, Paragraph”.

13/2 {*AI95-00285-01*} {*format_effector*} format_effector
The characters whose code positions are 16#09# (CHARACTER TABULATION), 16#0A# (LINE FEED), 16#0B# (LINE TABULATION), 16#0C# (FORM FEED), 16#0D# (CARRIAGE RETURN), 16#85# (NEXT LINE), and the characters in categories separator_line and separator_paragraph. {*control character: See also format_effector*}

13.a/2 **Discussion:** ISO/IEC 10646:2003 does not define the names of control characters, but rather refers to the names defined by ISO/IEC 6429:1992. These are the names that we use here.

13.1/2 {*AI95-00285-01*} {*other_control*} other_control
Any character whose General Category is defined to be “Other, Control”, and which is not defined to be a format_effector.

13.2/2 {*AI95-00285-01*} {*other_private_use*} other_private_use
Any character whose General Category is defined to be “Other, Private Use”.

13.3/2 {*AI95-00285-01*} {*other_surrogate*} other_surrogate
Any character whose General Category is defined to be “Other, Surrogate”.

14/2 {*AI95-00285-01*} {*AI95-00395-01*} {*graphic_character*} graphic_character
Any character that is not in the categories other_control, other_private_use, other_surrogate, format_effector, and whose relative code position in its plane is neither 16#FFFE# nor 16#FFFF#.

This paragraph was deleted. 14.a/2

Discussion: {*AI95-00285-01*} We considered basing the definition of lexical elements on Annex A of ISO/IEC TR 10176 (4th edition), which lists the characters which should be supported in identifiers for all programming languages, but we finally decided against this option. Note that it is not our intent to diverge from ISO/IEC TR 10176, except to the extent that ISO/IEC TR 10176 itself diverges from ISO/IEC 10646:2003 (which is the case at the time of this writing [January 2005]). 14.b/2

More precisely, we intend to align strictly with ISO/IEC 10646:2003. It must be noted that ISO/IEC TR 10176 is a Technical Report while ISO/IEC 10646:2003 is a Standard. If one has to make a choice, one should conform with the Standard rather than with the Technical Report. And, it turns out that one *must* make a choice because there are important differences between the two: 14.c/2

- ISO/IEC TR 10176 is still based on ISO/IEC 10646:2000 while ISO/IEC 10646:2003 has already been published for a year. We cannot afford to delay the adoption of our amendment until ISO/IEC TR 10176 has been revised. 14.d/2
- There are considerable differences between the two editions of ISO/IEC 10646, notably in supporting characters beyond the BMP (this might be significant for some languages, e.g. Korean). 14.e/2
- ISO/IEC TR 10176 does not define case conversion tables, which are essential for a case-insensitive language like Ada. To get case conversion tables, we would have to reference either ISO/IEC 10646:2003 or Unicode, or we would have to invent our own. 14.f/2

For the purpose of defining the lexical elements of the language, we need character properties like categorization, as well as case conversion tables. These are mentioned in ISO/IEC 10646:2003 as useful for implementations, with a reference to Unicode. Machine-readable tables are available on the web at URLs: 14.g/2

14.h/2
```
http://www.unicode.org/Public/4.0-Update/UnicodeData-4.0.0.txt
http://www.unicode.org/Public/4.0-Update/CaseFolding-4.0.0.txt
```

with an explanatory document found at URL: 14.i/2

14.j/2
```
http://www.unicode.org/Public/4.0-Update/UCD-4.0.0.html
```

The actual text of the standard only makes specific references to the corresponding clauses of ISO/IEC 10646:2003, not to Unicode. 14.k/2

{*AI95-00285-01*} The following names are used when referring to certain characters (the first name is that given in ISO/IEC 10646:2003): {*quotation mark*} {*number sign*} {*ampersand*} {*apostrophe*} {*tick*} {*left parenthesis*} {*right parenthesis*} {*asterisk*} {*multiply*} {*plus sign*} {*comma*} {*hyphen-minus*} {*minus*} {*full stop*} {*dot*} {*point*} {*solidus*} {*divide*} {*colon*} {*semicolon*} {*less-than sign*} {*equals sign*} {*greater-than sign*} {*low line*} {*underline*} {*vertical line*} {*exclamation point*} {*percent sign*} 15/2

Discussion: {*AI95-00285-01*} {*graphic symbols*} {*glyphs*} This table serves to show the correspondence between ISO/IEC 10646:2003 names and the graphic symbols (glyphs) used in this International Standard. These are the characters that play a special role in the syntax of Ada. 15.a/2

graphic symbol	name	graphic symbol	name
"	quotation mark	:	colon
#	number sign	;	semicolon
&	ampersand	<	less-than sign
'	apostrophe, tick	=	equals sign
(	left parenthesis	>	greater-than sign
)	right parenthesis	_	low line, underline
*	asterisk, multiply	|	vertical line
+	plus sign	/	solidus, divide
,	comma	!	exclamation point
–	hyphen-minus, minus	%	percent sign
.	full stop, dot, point		

Implementation Permissions

This paragraph was deleted.{*AI95-00285-01*} 16/2

NOTES

17/2 1 {*AI95-00285-01*} The characters in categories other_control, other_private_use, and other_surrogate are only allowed in comments.

18 2 The language does not specify the source representation of programs.

18.a/2 **Discussion:** Any source representation is valid so long as the implementer can produce an (information-preserving) algorithm for translating both directions between the representation and the standard character set. (For example, every character in the standard character set has to be representable, even if the output devices attached to a given computer cannot print all of those characters properly.) From a practical point of view, every implementer will have to provide some way to process the ACATS. It is the intent to allow source representations, such as parse trees, that are not even linear sequences of characters. It is also the intent to allow different fonts: reserved words might be in bold face, and that should be irrelevant to the semantics.

Extensions to Ada 83

18.b {*extensions to Ada 83*} Ada 95 allows 8-bit and 16-bit characters, as well as implementation-specified character sets.

Wording Changes from Ada 83

18.c/2 {*AI95-00285-01*} The syntax rules in this clause are modified to remove the emphasis on basic characters vs. others. (In this day and age, there is no need to point out that you can write programs without using (for example) lower case letters.) In particular, character (representing all characters usable outside comments) is added, and basic_graphic_character, other_special_character, and basic_character are removed. Special_character is expanded to include Ada 83's other_special_character, as well as new 8-bit characters not present in Ada 83. Ada 2005 removes special_character altogether; we want to stick to ISO/IEC 10646:2003 character classifications. Note that the term “basic letter” is used in A.3, “Character Handling” to refer to letters without diacritical marks.

18.d/2 {*AI95-00285-01*} Character names now come from ISO/IEC 10646:2003.

18.e/2 *This paragraph was deleted.*{*AI95-00285-01*}

Extensions to Ada 95

18.f/2 {*AI95-00285-01*} {*AI95-00395-01*} {*extensions to Ada 95*} Program text can use most characters defined by ISO-10646:2003. This clause has been rewritten to use the categories defined in that Standard. This should ease programming in languages other than English.

2.2 Lexical Elements, Separators, and Delimiters

Static Semantics

1 {*text of a program*} The text of a program consists of the texts of one or more compilations. {*lexical element*} {*token: See lexical element*} The text of each compilation is a sequence of separate *lexical elements*. Each lexical element is formed from a sequence of characters, and is either a delimiter, an identifier, a reserved word, a numeric_literal, a character_literal, a string_literal, or a comment. The meaning of a program depends only on the particular sequences of lexical elements that form its compilations, excluding comments.

2/2 {*AI95-00285-01*} The text of a compilation is divided into {*line*} *lines*. {*end of a line*} In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than the character whose code position is 16#09# (CHARACTER TABULATION) signifies at least one end of line.

2.a **Implementation defined:** The representation for an end of line.

3/2 {*AI95-00285-01*} {*separator*} [In some cases an explicit *separator* is required to separate adjacent lexical elements.] A separator is any of a separator_space, a format_effector, or the end of a line, as follows:

- 4/2 {*AI95-00285-01*} A separator_space is a separator except within a comment, a string_literal, or a character_literal.
- 5/2 {*AI95-00285-01*} The character whose code position is 16#09# (CHARACTER TABULATION) is a separator except within a comment.

- The end of a line is always a separator. 6

One or more separators are allowed between any two adjacent lexical elements, before the first of each compilation, or after the last. At least one separator is required between an identifier, a reserved word, or a numeric_literal and an adjacent identifier, reserved word, or numeric_literal. 7

{*AI95-00285-01*} {*delimiter*} A *delimiter* is either one of the following characters: 8/2

& ' ( ) * + , – . / : ; < = > | 9

{*compound delimiter*} or one of the following *compound delimiters* each composed of two adjacent special characters 10

=> .. ** := /= >= <= << >> <> 11

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string_literal, character_literal, or numeric_literal. 12

The following names are used when referring to compound delimiters: 13

delimiter	name
=>	arrow
..	double dot
**	double star, exponentiate
:=	assignment (pronounced: “becomes”)
/=	inequality (pronounced: “not equal”)
>=	greater than or equal
<=	less than or equal
<<	left label bracket
>>	right label bracket
<>	box

Implementation Requirements

An implementation shall support lines of at least 200 characters in length, not counting any characters used to signify the end of a line. An implementation shall support lexical elements of at least 200 characters in length. The maximum supported line length and lexical element length are implementation defined. 14

Implementation defined: Maximum supported line length and lexical element length. 14.a

Discussion: From URG recommendation. 14.b

Wording Changes from Ada 95

{*AI95-00285-01*} The wording was updated to use the new character categories defined in the preceding clause. 14.c/2

2.3 Identifiers

Identifiers are used as names. 1

Syntax

2/2 {*AI95-00285-01*} {*AI95-00395-01*} identifier ::=
  identifier_start {identifier_start | identifier_extend}

3/2 {*AI95-00285-01*} {*AI95-00395-01*} identifier_start ::=
   letter_uppercase
  | letter_lowercase
  | letter_titlecase
  | letter_modifier
  | letter_other
  | number_letter

3.1/2 {*AI95-00285-01*} {*AI95-00395-01*} identifier_extend ::=
   mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector
  | other_format

4/2 {*AI95-00395-01*} After eliminating the characters in category other_format, an identifier shall not contain two consecutive characters in category punctuation_connector, or end with a character in that category.

4.a/2 **Reason:** This rule was stated in the syntax in Ada 95, but that has gotten too complex in Ada 2005. Since other_format characters usually do not display, we do not want to count them as separating two underscores.

Static Semantics

5/2 {*AI95-00285-01*} Two identifiers are considered the same if they consist of the same sequence of characters after applying the following transformations (in this order):

5.1/2 • {*AI95-00285-01*} The characters in category other_format are eliminated.

5.2/2 • {*AI95-00285-01*} {*AI95-00395-01*} The remaining sequence of characters is converted to upper case. {*case insensitive*}

5.3/2 {*AI95-00395-01*} After applying these transformations, an identifier shall not be identical to a reserved word (in upper case).

5.b/2 **Implementation Note:** We match the reserved words after doing these transformations so that the rules for identifiers and reserved words are the same. (This allows other_format characters, which usually don't display, in a reserved word without changing it to an identifier.) Since a compiler usually will lexically process identifiers and reserved words the same way (often with the same code), this will prevent a lot of headaches.

5.c/2 **Ramification:** The rules for reserved words differ in one way: they define case conversion on letters rather than sequences. This means that some unusual sequences are neither identifiers nor reserved words. For instance, “ıf” and “acceß” have upper case conversions of “IF” and “ACCESS” respectively. These are not identifiers, because the transformed values are identical to a reserved word. But they are not reserved words, either, because the original values do not match any reserved word as defined or with any number of characters of the reserved word in upper case. Thus, these odd constructions are just illegal, and should not appear in the source of a program.

Implementation Permissions

6 In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers[, to accommodate local conventions].

6.a/2 **Discussion:** {*AI95-00285-01*} For instance, in most languages, the uppercase equivalent of LATIN SMALL LETTER I (a lower case letter with a dot above) is LATIN CAPITAL LETTER I (an upper case letter without a dot above). In Turkish, though, LATIN SMALL LETTER I and LATIN SMALL LETTER DOTLESS I are two distinct letters, so the upper case equivalent of LATIN SMALL LETTER I is LATIN CAPITAL LETTER I WITH DOT ABOVE, and the upper case equivalent of LATIN SMALL LETTER DOTLESS I is LATIN CAPITAL LETTER I. Take for instance the following identifier (which is the name of a city on the Tigris river in Eastern Anatolia):

6.b/2
```
diyarbakır -- The first i is dotted, the second isn't.
```

Locale-independent conversion to upper case results in: 6.c/2

```
DIYARBAKIR -- Both Is are dotless.
```
6.d/2

This means that the four following sequences of characters represent the same identifier, even though for a locutor of Turkish they would probably be considered distinct words: 6.e/2

```
diyarbakir
diyarbakır
dıyarbakir
dıyarbakır
```
6.f/2

An implementation targeting the Turkish market is allowed (in fact, expected) to provide a nonstandard mode where case folding is appropriate for Turkish. This would cause the original identifier to be converted to: 6.g/2

```
DİYARBAKIR -- The first I is dotted, the second isn't.
```
6.h/2

and the four sequences of characters shown above would represent four distinct identifiers. 6.i/2

Lithuanian and Azeri are two other languages that present similar idiosyncrasies. 6.j/2

NOTES
3 {*AI95-00285-01*} Identifiers differing only in the use of corresponding upper and lower case letters are considered the same. 6.1/2

Examples

Examples of identifiers: 7

```
{AI95-00433-01} Count      X    Get_Symbol   Ethelyn   Marion
Snobol_4   X1   Page_Count   Store_Next_Item
Πλάτων        -- Plato
Чайковский   -- Tchaikovsky
θ  φ          -- Angles
```
8/2

Wording Changes from Ada 83

We no longer include reserved words as identifiers. This is not a language change. In Ada 83, identifier included reserved words. However, this complicated several other rules (for example, regarding implementation-defined attributes and pragmas, etc.). We now explicitly allow certain reserved words for attribute designators, to make up for the loss. 8.a

Ramification: Because syntax rules are relevant to overload resolution, it means that if it looks like a reserved word, it is not an identifier. As a side effect, implementations cannot use reserved words as implementation-defined attributes or pragma names. 8.b

Extensions to Ada 95

{*AI95-00285-01*} {*extensions to Ada 95*} An identifier can use any letter defined by ISO-10646:2003, along with several other categories. This should ease programming in languages other than English. 8.c/2

2.4 Numeric Literals

{*literal (numeric)*} There are two kinds of numeric_literals, *real literals* and *integer literals*. {*real literal*} A real literal is a numeric_literal that includes a point; {*integer literal*} an integer literal is a numeric_literal without a point. 1

Syntax

numeric_literal ::= decimal_literal | based_literal 2

NOTES
4 The type of an integer literal is *universal_integer*. The type of a real literal is *universal_real*. 3

2.4.1 Decimal Literals

{*literal (decimal)*} A decimal_literal is a numeric_literal in the conventional decimal notation (that is, the base is ten). 1

Syntax

2 decimal_literal ::= numeral [.numeral] [exponent]

3 numeral ::= digit {[underline] digit}

4 exponent ::= E [+] numeral | E – numeral

4.1/2 {*AI95-00285-01*} digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

5 An exponent for an integer literal shall not have a minus sign.

5.a **Ramification:** Although this rule is in this subclause, it applies also to the next subclause.

Static Semantics

6 An underline character in a numeric_literal does not affect its meaning. The letter E of an exponent can be written either in lower case or in upper case, with the same meaning.

6.a **Ramification:** Although these rules are in this subclause, they apply also to the next subclause.

7 An exponent indicates the power of ten by which the value of the decimal_literal without the exponent is to be multiplied to obtain the value of the decimal_literal with the exponent.

Examples

8 *Examples of decimal literals:*

9
```
12        0      1E6    123_456     -- integer literals

12.0      0.0    0.456  3.14159_26  -- real literals
```

Wording Changes from Ada 83

9.a We have changed the syntactic category name integer to be numeral. We got this idea from ACID. It avoids the confusion between this and integers. (Other places don't offer similar confusions. For example, a string_literal is different from a string.)

2.4.2 Based Literals

1 [{*literal (based)*} {*binary literal*} {*base 2 literal*} {*binary (literal)*} {*octal literal*} {*base 8 literal*} {*octal (literal)*} {*hexadecimal literal*} {*base 16 literal*} {*hexadecimal (literal)*} A based_literal is a numeric_literal expressed in a form that specifies the base explicitly.]

Syntax

2 based_literal ::=
  base # based_numeral [.based_numeral] # [exponent]

3 base ::= numeral

4 based_numeral ::=
  extended_digit {[underline] extended_digit}

5 extended_digit ::= digit | A | B | C | D | E | F

Legality Rules

6 {*base*} The *base* (the numeric value of the decimal numeral preceding the first #) shall be at least two and at most sixteen. The extended_digits A through F represent the digits ten through fifteen, respectively. The value of each extended_digit of a based_literal shall be less than the base.

Static Semantics

7 The conventional meaning of based notation is assumed. An exponent indicates the power of the base by which the value of the based_literal without the exponent is to be multiplied to obtain the value of the based_literal with the exponent. The base and the exponent, if any, are in decimal notation.

8 The extended_digits A through F can be written either in lower case or in upper case, with the same meaning.

Examples

9 *Examples of based literals:*

10
```
2#1111_1111#  16#FF#       016#0ff#    -- integer literals of value 255
16#E#E1       2#1110_0000#             -- integer literals of value 224
16#F.FF#E+2   2#1.1111_1111_1110#E11   -- real literals of value 4095.0
```

Wording Changes from Ada 83

10.a The rule about which letters are allowed is now encoded in BNF, as suggested by Mike Woodger. This is clearly more readable.

2.5 Character Literals

1 [A character_literal is formed by enclosing a graphic character between two apostrophe characters.]

Syntax

2 character_literal ::= 'graphic_character'

NOTES

3 5 A character_literal is an enumeration literal of a character type. See 3.5.2.

Examples

4 *Examples of character literals:*

5/2
```
{AI95-00433-01} 'A'     '*'     '''     ' '
'L'     'Л'     'Λ'     -- Various els.
'∞'     'א'             -- Big numbers - infinity and aleph.
```

Wording Changes from Ada 83

5.a The definitions of the values of literals are in Sections 3 and 4, rather than here, since it requires knowledge of types.

2.6 String Literals

1 [A string_literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets. They are used to represent operator_symbols (see 6.1), values of a string type (see 4.2), and array subaggregates (see 4.3.3). {*quoted string: See string_literal*}]

Syntax

2 string_literal ::= "{string_element}"

3 string_element ::= "" | *non_quotation_mark*_graphic_character

4 A string_element is either a pair of quotation marks (""), or a single graphic_character other than a quotation mark.

Static Semantics

5 {*sequence of characters (of a string_literal)*} The *sequence of characters* of a string_literal is formed from the sequence of string_elements between the bracketing quotation marks, in the given order, with a

string_element that is "" becoming a single quotation mark in the sequence of characters, and any other string_element being reproduced in the sequence.

6 {*null string literal*} A *null string literal* is a string_literal with no string_elements between the quotation marks.

NOTES

7 6 An end of line cannot appear in a string_literal.

7.1/2 7 {*AI95-00285-01*} No transformation is performed on the sequence of characters of a string_literal.

Examples

8 *Examples of string literals:*

9/2
```
{AI95-00433-01} "Message of the day:"

""                          -- a null string literal
" "   "A"   """"            -- three string literals of length 1

"Characters such as $, %, and } are allowed in string literals"
"Archimedes said ""Εύρηκα"""
"Volume of cylinder (πr²h) = "
```

Wording Changes from Ada 83

9.a The wording has been changed to be strictly lexical. No mention is made of string or character values, since string_literals are also used to represent operator_symbols, which don't have a defined value.

9.b The syntax is described differently.

Wording Changes from Ada 95

9.c/2 {*AI95-00285-01*} We explicitly say that the characters of a string_literal should be used as is. In particular, no normalization or folding should be performed on a string_literal.

2.7 Comments

1 A comment starts with two adjacent hyphens and extends up to the end of the line.

Syntax

2 comment ::= --{*non_end_of_line*_character}

3 A comment may appear on any line of a program.

Static Semantics

4 The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the meaning of a program; their sole purpose is the enlightenment of the human reader.

Examples

5 *Examples of comments:*

6
```
--  the last sentence above echoes the Algol 68 report

end;    --  processing of Line is complete

--  a long comment may be split onto
--  two or more consecutive lines

----------------  the first two hyphens start the comment
```

2.8 Pragmas

{*Pragma*} [Glossary Entry]A pragma is a compiler directive. There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation-defined) pragmas. 1

Syntax

pragma ::=
  pragma identifier [(pragma_argument_association {, pragma_argument_association})]; 2

pragma_argument_association ::=
    [*pragma_argument*_identifier =>] name
  | [*pragma_argument*_identifier =>] expression 3

In a pragma, any pragma_argument_associations without a *pragma_argument*_identifier shall precede any associations with a *pragma_argument*_identifier. 4

Pragmas are only allowed at the following places in a program: 5

- After a semicolon delimiter, but not within a formal_part or discriminant_part. 6
- At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "statement", "clause", or "alternative", or one of the syntactic categories variant or exception_handler; but not in place of such a construct. Also at any place where a compilation_unit would be allowed. 7

Additional syntax rules and placement restrictions exist for specific pragmas. 8

Discussion: The above rule is written in text, rather than in BNF; the syntactic category pragma is not used in any BNF syntax rule. 8.a

Ramification: A pragma is allowed where a generic_formal_parameter_declaration is allowed. 8.b

{*name (of a pragma)*} {*pragma name*} The *name* of a pragma is the identifier following the reserved word **pragma**. {*pragma argument*} {*argument of a pragma*} The name or expression of a pragma_argument_association is a *pragma argument*. 9

To be honest: {*AI95-00284-02*} For compatibility with Ada 83, the name of a pragma may also be "**interface**", which is not an identifier (because it is a reserved word). See J.12. 9.a/2

{*identifier specific to a pragma*} {*pragma, identifier specific to*} An *identifier specific to a pragma* is an identifier that is used in a pragma argument with special meaning for that pragma. 10

To be honest: Whenever the syntax rules for a given pragma allow "identifier" as an argument of the pragma, that identifier is an identifier specific to that pragma. 10.a

Static Semantics

If an implementation does not recognize the name of a pragma, then it has no effect on the semantics of the program. Inside such a pragma, the only rules that apply are the Syntax Rules. 11

To be honest: This rule takes precedence over any other rules that imply otherwise. 11.a

Ramification: Note well: this rule applies only to pragmas whose name is not recognized. If anything else is wrong with a pragma (at compile time), the pragma is illegal. This is true whether the pragma is language defined or implementation defined. 11.b

For example, an expression in an unrecognized pragma does not cause freezing, even though the rules in 13.14, "Freezing Rules" say it does; the above rule overrules those other rules. On the other hand, an expression in a recognized pragma causes freezing, even if this makes something illegal. 11.c

For another example, an expression that would be ambiguous is not illegal if it is inside an unrecognized pragma. 11.d

Note, however, that implementations have to recognize **pragma** Inline(Foo) and freeze things accordingly, even if they choose to never do inlining. 11.e

11.f Obviously, the contradiction needs to be resolved one way or the other. The reasons for resolving it this way are: The implementation is simple — the compiler can just ignore the pragma altogether. The interpretation of constructs appearing inside implementation-defined pragmas is implementation defined. For example: "**pragma** Mumble(X);". If the current implementation has never heard of Mumble, then it doesn't know whether X is a name, an expression, or an identifier specific to the pragma Mumble.

11.g **To be honest:** The syntax of individual pragmas overrides the general syntax for pragma.

11.h **Ramification:** Thus, an identifier specific to a pragma is not a name, syntactically; if it were, the visibility rules would be invoked, which is not what we want.

11.i This also implies that named associations do not allow one to give the arguments in an arbitrary order — the order given in the syntax rule for each individual pragma must be obeyed. However, it is generally possible to leave out earlier arguments when later ones are given; for example, this is allowed by the syntax rule for pragma Import (see B.1, "Interfacing Pragmas"). As for subprogram calls, positional notation precedes named notation.

11.j Note that Ada 83 had no pragmas for which the order of named associations mattered, since there was never more than one argument that allowed named associations.

11.k **To be honest:** The interpretation of the arguments of implementation-defined pragmas is implementation defined. However, the syntax rules have to be obeyed.

Dynamic Semantics

12 {*execution (pragma)* [partial]} {*elaboration (pragma)* [partial]} Any pragma that appears at the place of an executable construct is executed. Unless otherwise specified for a particular pragma, this execution consists of the evaluation of each evaluable pragma argument in an arbitrary order.

12.a **Ramification:** For a pragma that appears at the place of an elaborable construct, execution is elaboration.

12.b An identifier specific to a pragma is neither a name nor an expression — such identifiers are not evaluated (unless an implementation defines them to be evaluated in the case of an implementation-defined pragma).

12.c The "unless otherwise specified" part allows us (and implementations) to make exceptions, so a pragma can contain an expression that is not evaluated. Note that pragmas in type_definitions may contain expressions that depend on discriminants.

12.d When we wish to define a pragma with some run-time effect, we usually make sure that it appears in an executable context; otherwise, special rules are needed to define the run-time effect and when it happens.

Implementation Requirements

13 The implementation shall give a warning message for an unrecognized pragma name.

13.a **Ramification:** An implementation is also allowed to have modes in which a warning message is suppressed, or in which the presence of an unrecognized pragma is a compile-time error.

Implementation Permissions

14 An implementation may provide implementation-defined pragmas; the name of an implementation-defined pragma shall differ from those of the language-defined pragmas.

14.a **Implementation defined:** Implementation-defined pragmas.

14.b **Ramification:** The semantics of implementation-defined pragmas, and any associated rules (such as restrictions on their placement or arguments), are, of course, implementation defined. Implementation-defined pragmas may have run-time effects.

15 An implementation may ignore an unrecognized pragma even if it violates some of the Syntax Rules, if detecting the syntax error is too complex.

15.a **Reason:** Many compilers use extra post-parsing checks to enforce the syntax rules, since the Ada syntax rules are not LR(k) (for any k). (The grammar is ambiguous, in fact.) This paragraph allows them to ignore an unrecognized pragma, without having to perform such post-parsing checks.

Implementation Advice

16 Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas are removed from a working program, the program should still be legal, and should still have the same semantics.

Implementation Advice: Implementation-defined pragmas should have no semantic effect for error-free programs. 16.a.1/2

Ramification: Note that “semantics” is not the same as “effect;” as explained in 1.1.3, the semantics defines a set of possible effects. 16.a

Note that adding a pragma to a program might cause an error (either at compile time or at run time). On the other hand, if the language-specified semantics for a feature are in part implementation defined, it makes sense to support pragmas that control the feature, and that have real semantics; thus, this paragraph is merely a recommendation. 16.b

Normally, an implementation should not define pragmas that can make an illegal program legal, except as follows: 17

- A pragma used to complete a declaration, such as a pragma Import; 18
- A pragma used to configure the environment by adding, removing, or replacing library_items. 19

Implementation Advice: Implementation-defined pragmas should not make an illegal program legal, unless they complete a declaration or configure the library_items in an environment. 19.a.1/2

Ramification: For example, it is OK to support Interface, System_Name, Storage_Unit, and Memory_Size pragmas for upward compatibility reasons, even though all of these pragmas can make an illegal program legal. (The latter three can affect legality in a rather subtle way: They affect the value of named numbers in System, and can therefore affect the legality in cases where static expressions are required.) 19.a

On the other hand, adding implementation-defined pragmas to a legal program can make it illegal. For example, a common kind of implementation-defined pragma is one that promises some property that allows more efficient code to be generated. If the promise is a lie, it is best if the user gets an error message. 19.b

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} In Ada 83, “bad” pragmas are ignored. In Ada 95, they are illegal, except in the case where the name of the pragma itself is not recognized by the implementation. 19.c

Extensions to Ada 83

{*extensions to Ada 83*} Implementation-defined pragmas may affect the legality of a program. 19.d

Wording Changes from Ada 83

Implementation-defined pragmas may affect the run-time semantics of the program. This was always true in Ada 83 (since it was not explicitly forbidden by RM83), but it was not clear, because there was no definition of “executing” or “elaborating” a pragma. 19.e

Syntax

The forms of List, Page, and Optimize pragmas are as follows: 20

pragma List(identifier); 21

pragma Page; 22

pragma Optimize(identifier); 23

[Other pragmas are defined throughout this International Standard, and are summarized in Annex L.] 24

Ramification: The language-defined pragmas are supported by every implementation, although “supporting” some of them (for example, Inline) requires nothing more than checking the arguments, since they act only as advice to the implementation. 24.a

Static Semantics

A pragma List takes one of the identifiers On or Off as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a List pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing. 25

A pragma Page is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing). 26

27 A pragma Optimize takes one of the identifiers Time, Space, or Off as the single argument. This pragma is allowed anywhere a pragma is allowed, and it applies until the end of the immediately enclosing declarative region, or for a pragma at the place of a compilation_unit, to the end of the compilation. It gives advice to the implementation as to whether time or space is the primary optimization criterion, or that optional optimizations should be turned off. [It is implementation defined how this advice is followed.]

27.a **Implementation defined:** Effect of pragma Optimize.

27.b **Discussion:** For example, a compiler might use Time vs. Space to control whether generic instantiations are implemented with a macro-expansion model, versus a shared-generic-body model.

27.c We don't define what constitutes an “optimization” — in fact, it cannot be formally defined in the context of Ada. One compiler might call something an optional optimization, whereas another compiler might consider that same thing to be a normal part of code generation. Thus, the programmer cannot rely on this pragma having any particular portable effect on the generated code. Some compilers might even ignore the pragma altogether.

Examples

28 *Examples of pragmas:*

29/2
```
{AI95-00433-01} pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Inline(Set_Mask); -- generate code for Set_Mask inline
pragma Import(C, Put_Char, External_Name => "putchar"); -- import C putchar function
```

Extensions to Ada 83

29.a {*extensions to Ada 83*} The Optimize pragma now allows the identifier Off to request that normal optimization be turned off.

29.b An Optimize pragma may appear anywhere pragmas are allowed.

Wording Changes from Ada 83

29.c We now describe the pragmas Page, List, and Optimize here, to act as examples, and to remove the normative material from Annex L, “Language-Defined Pragmas”, so it can be entirely an informative annex.

Wording Changes from Ada 95

29.d/2 {*AI95-00433-01*} Updated the example of named pragma parameters, because the second parameter of pragma Suppress is obsolescent.

2.9 Reserved Words

Syntax

1/1 *This paragraph was deleted.*

2/2 {*AI95-00284-02*} {*AI95-00395-01*} {*reserved word*} The following are the *reserved words*. Within a program, some or all of the letters of a reserved word may be in upper case, and one or more characters in category other_format may be inserted within or at the end of the reserved word.

2.a **Discussion:** Reserved words have special meaning in the syntax. In addition, certain reserved words are used as attribute names.

2.b The syntactic category identifier no longer allows reserved words. We have added the few reserved words that are legal explicitly to the syntax for attribute_reference. Allowing identifier to include reserved words has been a source of confusion for some users, and differs from the way they are treated in the C and Pascal language definitions.

abort	**else**	**new**	**return**
abs	**elsif**	**not**	**reverse**
abstract	**end**	**null**	**select**
accept	**entry**	**of**	**separate**
access	**exception**	**or**	**subtype**
aliased	**exit**	**others**	**synchronized**
all	**for**	**out**	**tagged**
and	**function**	**overriding**	**task**
array	**generic**	**package**	**terminate**
at	**goto**	**pragma**	**then**
begin	**if**	**private**	**type**
body	**in**	**procedure**	**until**
case	**interface**	**protected**	**use**
constant	**is**	**raise**	**when**
declare	**limited**	**range**	**while**
delay	**loop**	**record**	**with**
delta	**mod**	**rem**	**xor**
digits		**renames**	
do		**requeue**	

NOTES

8 The reserved words appear in **lower case boldface** in this International Standard, except when used in the designator of an attribute (see 4.1.4). Lower case boldface is also used for a reserved word in a string_literal used as an operator_symbol. This is merely a convention — programs may be written in whatever typeface is desired and available. 3

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} The following words are not reserved in Ada 83, but are reserved in Ada 95: **abstract**, **aliased**, **protected**, **requeue**, **tagged**, **until**. 3.a

Wording Changes from Ada 83

The clause entitled “Allowed Replacements of Characters” has been moved to Annex J, “Obsolescent Features”. 3.b

Incompatibilities With Ada 95

{*AI95-00284-02*} {*incompatibilities with Ada 95*} The following words are not reserved in Ada 95, but are reserved in Ada 2005: **interface**, **overriding**, **synchronized**. A special allowance is made for **pragma** Interface (see J.12). Uses of these words as identifiers will need to be changed, but we do not expect them to be common. 3.c/2

Wording Changes from Ada 95

{*AI95-00395-01*} The definition of upper case equivalence has been modified to allow identifiers using all of the characters of ISO 10646. This change has no effect on the character sequences that are reserved words, but does make some unusual sequences of characters illegal. 3.d/2

Section 3: Declarations and Types

1 This section describes the types in the language and the rules for declaring constants, variables, and named numbers.

3.1 Declarations

1 {*entity* [partial]} The language defines several kinds of named *entities* that are declared by declarations. {*name* [partial]} The entity's *name* is defined by the declaration, usually by a defining_identifier, but sometimes by a defining_character_literal or defining_operator_symbol.

2 There are several forms of declaration. A basic_declaration is a form of declaration defined as follows.

Syntax

3/2
```
{AI95-00348-01} basic_declaration ::=
    type_declaration               | subtype_declaration
  | object_declaration             | number_declaration
  | subprogram_declaration         | abstract_subprogram_declaration
  | null_procedure_declaration     | package_declaration
  | renaming_declaration           | exception_declaration
  | generic_declaration            | generic_instantiation
```

4
```
defining_identifier ::= identifier
```

Static Semantics

5 {*Declaration*} [Glossary Entry]A *declaration* is a language construct that associates a name with (a view of) an entity. {*explicit declaration*} {*implicit declaration*} A declaration may appear explicitly in the program text (an *explicit* declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an *implicit* declaration).

> 5.a **Discussion:** An implicit declaration generally declares a predefined or inherited operation associated with the definition of a type. This term is used primarily when allowing explicit declarations to override implicit declarations, as part of a type declaration.

6/2 {*AI95-00318-02*} {*declaration*} Each of the following is defined to be a declaration: any basic_declaration; an enumeration_literal_specification; a discriminant_specification; a component_declaration; a loop_parameter_specification; a parameter_specification; a subprogram_body; an entry_declaration; an entry_index_specification; a choice_parameter_specification; a generic_formal_parameter_declaration. In addition, an extended_return_statement is a declaration of its defining_identifier.

> 6.a **Discussion:** This list (when basic_declaration is expanded out) contains all syntactic categories that end in "_declaration" or "_specification", except for program unit _specifications. Moreover, it contains subprogram_body. A subprogram_body is a declaration, whether or not it completes a previous declaration. This is a bit strange, subprogram_body is not part of the syntax of basic_declaration or library_unit_declaration. A renaming-as-body is considered a declaration. An accept_statement is not considered a declaration. Completions are sometimes declarations, and sometimes not.

7 {*view*} {*definition*} All declarations contain a *definition* for a *view* of an entity. A view consists of an identification of the entity (the entity *of* the view), plus view-specific characteristics that affect the use of the entity through that view (such as mode of access to an object, formal parameter names and defaults for a subprogram, or visibility to components of a type). In most cases, a declaration also contains the definition for the entity itself (a renaming_declaration is an example of a declaration that does not define a new entity, but instead defines a view of an existing entity (see 8.5)).

7.a/2 **Glossary entry:** {*View*} A view of an entity reveals some or all of the properties of the entity. A single entity may have multiple views.

7.b **Discussion:** Most declarations define a view (of some entity) whose view-specific characteristics are unchanging for the life of the view. However, subtypes are somewhat unusual in that they inherit characteristics from whatever view of their type is currently visible. Hence, a subtype is not a *view* of a type; it is more of an indirect reference. By contrast, a private type provides a single, unchanging (partial) view of its full type.

8 {*Definition*} [Glossary Entry]

9 {*scope (informal definition)* [partial]} For each declaration, the language rules define a certain region of text called the *scope* of the declaration (see 8.2). Most declarations associate an identifier with a declared entity. Within its scope, and only there, there are places where it is possible to use the identifier to refer to the declaration, the view it defines, and the associated entity; these places are defined by the visibility rules (see 8.3). {*name (of (a view of) an entity)*} At such places the identifier is said to be a *name* of the entity (the direct_name or selector_name); {*denote (informal definition)* [partial]} the name is said to *denote* the declaration, the view, and the associated entity (see 8.6). {*declare*} The declaration is said to *declare* the name, the view, and in most cases, the entity itself.

10 As an alternative to an identifier, an enumeration literal can be declared with a character_literal as its name (see 3.5.1), and a function can be declared with an operator_symbol as its name (see 6.1).

11 {*defining name*} The syntax rules use the terms defining_identifier, defining_character_literal, and defining_operator_symbol for the defining occurrence of a name; these are collectively called *defining names*. {*usage name*} The terms direct_name and selector_name are used for usage occurrences of identifiers, character_literals, and operator_symbols. These are collectively called *usage names*.

11.a **To be honest:** The terms identifier, character_literal, and operator_symbol are used directly in contexts where the normal visibility rules do not apply (such as the identifier that appears after the **end** of a task_body). Analogous conventions apply to the use of designator, which is the collective term for identifier and operator_symbol.

Dynamic Semantics

12 {*execution* [distributed]} The process by which a construct achieves its run-time effect is called *execution*. {*elaboration* [distributed]} {*evaluation* [distributed]} This process is also called *elaboration* for declarations and *evaluation* for expressions. One of the terms execution, elaboration, or evaluation is defined by this International Standard for each construct that has a run-time effect.

12.a **Glossary entry:** {*Execution*} The process by which a construct achieves its run-time effect is called *execution*. {*elaboration*} {*evaluation*} Execution of a declaration is also called *elaboration*. Execution of an expression is also called *evaluation*.

12.b **To be honest:** The term elaboration is also used for the execution of certain constructs that are not declarations, and the term evaluation is used for the execution of certain constructs that are not expressions. For example, subtype_indications are elaborated, and ranges are evaluated.

12.c For bodies, execution and elaboration are both explicitly defined. When we refer specifically to the execution of a body, we mean the explicit definition of execution for that kind of body, not its elaboration.

12.d **Discussion:** Technically, "the execution of a declaration" and "the elaboration of a declaration" are synonymous. We use the term "elaboration" of a construct when we know the construct is elaborable. When we are talking about more arbitrary constructs, we use the term "execution". For example, we use the term "erroneous execution", to refer to any erroneous execution, including erroneous elaboration or evaluation.

12.e When we explicitly define evaluation or elaboration for a construct, we are implicitly defining execution of that construct.

12.f We also use the term "execution" for things like statements, which are executable, but neither elaborable nor evaluable. We considered using the term "execution" only for non-elaborable, non-evaluable constructs, and defining the term "action" to mean what we have defined "execution" to mean. We rejected this idea because we thought three terms that mean the same thing was enough — four would be overkill. Thus, the term "action" is used only informally in the standard (except where it is defined as part of a larger term, such as "protected action").

12.f.1/2 **Glossary entry:** {*Elaboration*} The process by which a declaration achieves its run-time effect is called elaboration. Elaboration is one of the forms of execution.

Glossary entry: {*Evaluation*} The process by which an expression achieves its run-time effect is called evaluation. Evaluation is one of the forms of execution. 12.f.2/2

To be honest: {*elaborable*} A construct is *elaborable* if elaboration is defined for it. {*evaluable*} A construct is *evaluable* if evaluation is defined for it. {*executable*} A construct is *executable* if execution is defined for it. 12.g

Discussion: Don't confuse "elaborable" with "preelaborable" (defined in 10.2.1). 12.h

{*AI95-00114-01*} Evaluation of an evaluable construct produces a result that is either a value, a denotation, or a range. The following are evaluable: expression; name prefix; range; entry_index_specification; and possibly discrete_range. The last one is curious — RM83 uses the term "evaluation of a discrete_range," but never defines it. One might presume that the evaluation of a discrete_range consists of the evaluation of the range or the subtype_indication, depending on what it is. But subtype_indications are not evaluated; they are elaborated. 12.i/2

Intuitively, an *executable* construct is one that has a defined run-time effect (which may be null). Since execution includes elaboration and evaluation as special cases, all elaborable and all evaluable constructs are also executable. Hence, most constructs in Ada are executable. An important exception is that the constructs inside a generic unit are not executable directly, but rather are used as a template for (generally) executable constructs in instances of the generic. 12.j

NOTES

1 {*declare*} At compile time, the declaration of an entity *declares* the entity. {*create*} At run time, the elaboration of the declaration *creates* the entity. 13

Ramification: Syntactic categories for declarations are named either *entity*_declaration (if they include a trailing semicolon) or *entity*_specification (if not). 13.a

{*entity*} The various kinds of named entities that can be declared are as follows: an object (including components and parameters), a named number, a type (the name always refers to its first subtype), a subtype, a subprogram (including enumeration literals and operators), a single entry, an entry family, a package, a protected or task unit (which corresponds to either a type or a single object), an exception, a generic unit, a label, and the name of a statement. 13.b

Identifiers are also associated with names of pragmas, arguments to pragmas, and with attributes, but these are not user-definable. 13.c

Wording Changes from Ada 83

The syntax rule for defining_identifier is new. It is used for the defining occurrence of an identifier. Usage occurrences use the direct_name or selector_name syntactic categories. Each occurrence of an identifier (or simple_name), character_literal, or operator_symbol in the Ada 83 syntax rules is handled as follows in Ada 95: 13.d

- It becomes a defining_identifier, defining_character_literal, or defining_operator_symbol (or some syntactic category composed of these), to indicate a defining occurrence; 13.e
- It becomes a direct_name, in usage occurrences where the usage is required (in Section 8) to be directly visible; 13.f
- It becomes a selector_name, in usage occurrences where the usage is required (in Section 8) to be visible but not necessarily directly visible; 13.g
- It remains an identifier, character_literal, or operator_symbol, in cases where the visibility rules do not apply (such as the designator that appears after the **end** of a subprogram_body). 13.h

For declarations that come in "two parts" (program unit declaration plus body, private or incomplete type plus full type, deferred constant plus full constant), we consider both to be defining occurrences. Thus, for example, the syntax for package_body uses defining_identifier after the reserved word **body**, as opposed to direct_name. 13.i

The defining occurrence of a statement name is in its implicit declaration, not where it appears in the program text. Considering the statement name itself to be the defining occurrence would complicate the visibility rules. 13.j

The phrase "visible by selection" is not used in Ada 95. It is subsumed by simply "visible" and the Name Resolution Rules for selector_names. 13.k

(Note that in Ada 95, a declaration is visible at all places where one could have used a selector_name, not just at places where a selector_name was actually used. Thus, the places where a declaration is directly visible are a subset of the places where it is visible. See Section 8 for details.) 13.l

We use the term "declaration" to cover _specifications that declare (views of) objects, such as parameter_specifications. In Ada 83, these are referred to as a "form of declaration," but it is not entirely clear that they are considered simply "declarations." 13.m

RM83 contains an incomplete definition of "elaborated" in this clause: it defines "elaborated" for declarations, declarative_parts, declarative_items and compilation_units, but "elaboration" is defined elsewhere for various other constructs. To make matters worse, Ada 95 has a different set of elaborable constructs. Instead of correcting the list, it is more maintainable to refer to the term "elaborable," which is defined in a distributed manner. 13.n

13.o RM83 uses the term “has no other effect” to describe an elaboration that doesn't do anything except change the state from not-yet-elaborated to elaborated. This was a confusing wording, because the answer to “other than what?” was to be found many pages away. In Ada 95, we change this wording to “has no effect” (for things that truly do nothing at run time), and “has no effect other than to establish that so-and-so can happen without failing the Elaboration_Check” (for things where it matters).

13.p We make it clearer that the term "execution" covers elaboration and evaluation as special cases. This was implied in RM83. For example, "erroneous execution" can include any execution, and RM83-9.4(3) has, "The task designated by any other task object depends on the master whose execution creates the task object;" the elaboration of the master's declarative_part is doing the task creation.

Wording Changes from Ada 95

13.q/2 {*AI95-00318-02*} Added extended_return_statement to the list of declarations.

13.r/2 {*AI95-00348-01*} Added null procedures (see 6.7) to the syntax.

3.2 Types and Subtypes

Static Semantics

1 {*type*} {*primitive operation* [partial]} A *type* is characterized by a set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. {*object* [partial]} An *object* of a given type is a run-time entity that contains (has) a value of the type.

1.a/2 **Glossary entry:** {*Type*} Each object has a type. A *type* has an associated set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. Types are grouped into *categories*. Most language-defined categories of types are also *classes* of types.

1.b/2 **Glossary entry:** {*Subtype*} A subtype is a type together with a constraint or null exclusion, which constrains the values of the subtype to satisfy a certain condition. The values of a subtype are a subset of the values of its type.

2/2 {*AI95-00442-01*} {*category (of types)*} {*class (of types)*} Types are grouped into *categories* of types. {*language-defined class (of types)*} There exist several *language-defined categories* of types (see NOTES below), reflecting the similarity of their values and primitive operations.{*language-defined category (of types)*} [Most categories of types form *classes* of types.] {*elementary type*} *Elementary* types are those whose values are logically indivisible; {*composite type*} {*component*} *composite* types are those whose values are composed of *component* values. {*aggregate: See also composite type*}

2.a/2 **Proof:** {*AI95-00442-01*} The formal definition of *category* and *class* is found in 3.4.

2.b/2 **Glossary entry:** {*Class (of types)*} {*closed under derivation*} A class is a set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class. The set of types of a class share common properties, such as their primitive operations.

2.b.1/2 **Glossary entry:** {*Category (of types)*} A category of types is a set of types with one or more common properties, such as primitive operations. A category of types that is closed under derivation is also known as a *class*.

2.c **Glossary entry:** {*Elementary type*} An elementary type does not have components.

2.d/2 **Glossary entry:** {*Composite type*} A composite type may have components.

2.e **Glossary entry:** {*Scalar type*} A scalar type is either a discrete type or a real type.

2.f **Glossary entry:** {*Access type*} An access type has values that designate aliased objects. Access types correspond to “pointer types” or “reference types” in some other languages.

2.g **Glossary entry:** {*Discrete type*} A discrete type is either an integer type or an enumeration type. Discrete types may be used, for example, in case_statements and as array indices.

2.h **Glossary entry:** {*Real type*} A real type has values that are approximations of the real numbers. Floating point and fixed point types are real types.

2.i **Glossary entry:** {*Integer type*} Integer types comprise the signed integer types and the modular types. A signed integer type has a base range that includes both positive and negative numbers, and has operations that may raise an exception when the result is outside the base range. A modular type has a base range whose lower bound is zero, and has operations with “wraparound” semantics. Modular types subsume what are called “unsigned types” in some other languages.

Glossary entry: {*Enumeration type*} An enumeration type is defined by an enumeration of its values, which may be named by identifiers or character literals. 2.j

Glossary entry: {*Character type*} A character type is an enumeration type whose values include characters. 2.k

Glossary entry: {*Record type*} A record type is a composite type consisting of zero or more named components, possibly of different types. 2.l

Glossary entry: {*Record extension*} A record extension is a type that extends another type by adding additional components. 2.m

Glossary entry: {*Array type*} An array type is a composite type whose components are all of the same type. Components are selected by indexing. 2.n

Glossary entry: {*Task type*} A task type is a composite type used to represent active entities which execute concurrently and which can communicate via queued task entries. The top-level task of a partition is called the environment task. 2.o/2

Glossary entry: {*Protected type*} A protected type is a composite type whose components are accessible only through one of its protected operations which synchronize concurrent access by multiple tasks. 2.p/2

Glossary entry: {*Private type*} A private type gives a view of a type that reveals only some of its properties. The remaining properties are provided by the full view given elsewhere. Private types can be used for defining abstractions that hide unnecessary details from their clients. 2.q/2

Glossary entry: {*Private extension*} A private extension is a type that extends another type, with the additional properties hidden from its clients. 2.r/2

Glossary entry: {*Incomplete type*} An incomplete type gives a view of a type that reveals only some of its properties. The remaining properties are provided by the full view given elsewhere. Incomplete types can be used for defining recursive data structures. 2.s/2

{*scalar type*} The elementary types are the *scalar* types (*discrete* and *real*) and the *access* types (whose values provide access to objects or subprograms). {*discrete type*} {*enumeration type*} Discrete types are either *integer* types or are defined by enumeration of their values (*enumeration* types). {*real type*} Real types are either *floating point* types or *fixed point* types. 3

{*AI95-00251-01*} {*AI95-00326-01*} The composite types are the *record* types, *record extensions*, *array* types, *interface* types, *task* types, and *protected* types. 4/2

This paragraph was deleted.{*AI95-00442-01*} 4.a/2

{*AI95-00326-01*} {*incomplete type*} {*private type*} {*private extension*} There can be multiple views of a type with varying sets of operations. [An *incomplete* type represents an incomplete view (see 3.10.1) of a type with a very restricted usage, providing support for recursive data structures. A *private* type or *private extension* represents a partial view (see 7.3) of a type, providing support for data abstraction. The full view (see 3.2.1) of a type represents its complete definition.] An incomplete or partial view is considered a composite type[, even if the full view is not]. 4.1/2

Proof: The real definitions of the views are in the referenced clauses. 4.b/2

{*AI95-00326-01*} {*discriminant*} Certain composite types (and views thereof) have special components called *discriminants* whose values affect the presence, constraints, or initialization of other components. Discriminants can be thought of as parameters of the type. 5/2

{*AI95-00366-01*} {*subcomponent*} The term *subcomponent* is used in this International Standard in place of the term component to indicate either a component, or a component of another subcomponent. Where other subcomponents are excluded, the term component is used instead. {*part (of an object or value)*} Similarly, a *part* of an object or value is used to mean the whole object or value, or any set of its subcomponents. The terms component, subcomponent, and part are also applied to a type meaning the component, subcomponent, or part of objects and values of the type. 6/2

Discussion: The definition of “part” here is designed to simplify rules elsewhere. By design, the intuitive meaning of “part” will convey the correct result to the casual reader, while this formalistic definition will answer the concern of the compiler-writer. 6.a

6.b We use the term “part” when talking about the parent part, ancestor part, or extension part of a type extension. In contexts such as these, the part might represent an empty set of subcomponents (e.g. in a null record extension, or a nonnull extension of a null record). We also use “part” when specifying rules such as those that apply to an object with a “controlled part” meaning that it applies if the object as a whole is controlled, or any subcomponent is.

7/2 {*AI95-00231-01*} {*constraint* [partial]} The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* {*null constraint*} (the case of a *null constraint* that specifies no restriction is also included)[; the rules for which values satisfy a given kind of constraint are given in 3.5 for range_constraints, 3.6.1 for index_constraints, and 3.7.1 for discriminant_constraints]. The set of possible values for an object of an access type can also be subjected to a condition that excludes the null value (see 3.10).

8/2 {*AI95-00231-01*} {*AI95-00415-01*} {*subtype*} A *subtype* of a given type is a combination of the type, a constraint on values of the type, and certain attributes specific to the subtype. The given type is called the *type of the subtype*.{*type (of a subtype)*} {*subtype (type of)*} Similarly, the associated constraint is called the *constraint of the subtype*.{*constraint (of a subtype)*} {*subtype (constraint of)*} The set of values of a subtype consists of the values of its type that satisfy its constraint and any exclusion of the null value. {*belong (to a subtype)*} Such values *belong* to the subtype.{*values (belonging to a subtype)*} {*subtype (values belonging to)*}

8.a **Discussion:** We make a strong distinction between a type and its subtypes. In particular, a type is *not* a subtype of itself. There is no constraint associated with a type (not even a null one), and type-related attributes are distinct from subtype-specific attributes.

8.b **Discussion:** We no longer use the term "base type." All types were "base types" anyway in Ada 83, so the term was redundant, and occasionally confusing. In the RM95 we say simply "the type *of* the subtype" instead of "the base type of the subtype."

8.c **Ramification:** The value subset for a subtype might be empty, and need not be a proper subset.

8.d/2 **To be honest:** {*AI95-00442-01*} Any name of a category of types (such as “discrete”, “real”, or “limited”) is also used to qualify its subtypes, as well as its objects, values, declarations, and definitions, such as an “integer type declaration” or an “integer value.” In addition, if a term such as “parent subtype” or “index subtype” is defined, then the corresponding term for the type of the subtype is “parent type” or “index type.”

8.e **Discussion:** We use these corresponding terms without explicitly defining them, when the meaning is obvious.

9 {*constrained*} {*unconstrained*} {*constrained (subtype)*} {*unconstrained (subtype)*} A subtype is called an *unconstrained* subtype if its type has unknown discriminants, or if its type allows range, index, or discriminant constraints, but the subtype does not impose such a constraint; otherwise, the subtype is called a *constrained* subtype (since it has no unconstrained characteristics).

9.a **Discussion:** In an earlier version of Ada 9X, "constrained" meant "has a non-null constraint." However, we changed to this definition since we kept having to special case composite non-array/non-discriminated types. It also corresponds better to the (now obsolescent) attribute 'Constrained.

9.b For scalar types, “constrained” means “has a non-null constraint”. For composite types, in implementation terms, “constrained” means that the size of all objects of the subtype is the same, assuming a typical implementation model.

9.c Class-wide subtypes are always unconstrained.

NOTES

10/2 2 {*AI95-00442-01*} Any set of types can be called a “category” of types, and any set of types that is closed under derivation (see 3.4) can be called a “class” of types. However, only certain categories and classes are used in the description of the rules of the language — generally those that have their own particular set of primitive operations (see 3.2.3), or that correspond to a set of types that are matched by a given kind of generic formal type (see 12.5). {*language-defined class* [partial]} The following are examples of “interesting” *language-defined classes*: elementary, scalar, discrete, enumeration, character, boolean, integer, signed integer, modular, real, floating point, fixed point, ordinary fixed point, decimal fixed point, numeric, access, access-to-object, access-to-subprogram, composite, array, string, (untagged) record, tagged, task, protected, nonlimited. Special syntax is provided to define types in each of these classes. In addition to these classes, the following are examples of “interesting” *language-defined categories*: {*language-defined categories* [partial]} abstract, incomplete, interface, limited, private, record.

10.a **Discussion:** {*value*} A *value* is a run-time entity with a given type which can be assigned to an object of an appropriate subtype of the type. {*operation*} An *operation* is a program entity that operates on zero or more operands to produce an effect, or yield a result, or both.

Ramification: {*AI95-00442-01*} Note that a type's category (and class) depends on the place of the reference — a private type is composite outside and possibly elementary inside. It's really the *view* that is elementary or composite. Note that although private types are composite, there are some properties that depend on the corresponding full view — for example, parameter passing modes, and the constraint checks that apply in various places. 10.b/2

{*AI95-00345-01*} {*AI95-00442-01*} Every property of types forms a category, but not every property of types represents a class. For example, the set of all abstract types does not form a class, because this set is not closed under derivation. Similarly, the set of all interface types does not form a class. 10.c/2

{*AI95-00442-01*} The set of limited types does not form a class (since nonlimited types can inherit from limited interfaces), but the set of nonlimited types does. The set of tagged record types and the set of tagged private types do not form a class (because each of them can be extended to create a type of the other category); that implies that the set of record types and the set of private types also do not form a class (even though untagged record types and untagged private types do form a class). In all of these cases, we can talk about the category of the type; for instance, we can talk about the “category of limited types”.. 10.d/2

{*AI95-00442-01*} Normatively, the *language-defined classes* are those that are defined to be inherited on derivation by 3.4; other properties either aren't interesting or form categories, not classes. 10.e/2

{*AI95-00442-01*} These language-defined categories are organized like this: 11/2

- {*AI95-00345-01*} all types 12/2
    - elementary
        - scalar
            - discrete
                - enumeration
                    - character
                    - boolean
                    - other enumeration
                - integer
                    - signed integer
                    - modular integer
            - real
                - floating point
                - fixed point
                    - ordinary fixed point
                    - decimal fixed point
        - access
            - access-to-object
            - access-to-subprogram
    - composite
        - untagged
            - array
                - string
                - other array
            - record
            - task
            - protected
        - tagged (including interfaces)
            - nonlimited tagged record
            - limited tagged
                - limited tagged record
                - synchronized tagged
                    - tagged task
                    - tagged protected

{*AI95-00345-01*} {*AI95-00442-01*} There are other categories, such as “numeric” and “discriminated”, which represent other categorization dimensions, but do not fit into the above strictly hierarchical picture. 13/2

Discussion: {*AI95-00345-01*} {*AI95-00442-01*} Note that this is also true for some categories mentioned in the chart. The category “task” includes both untagged tasks and tagged tasks. Similarly for “protected”, “limited”, and “nonlimited” (note that limited and nonlimited are not shown for untagged composite types). 13.a.1/2

Wording Changes from Ada 83

This clause and its subclauses now precede the clause and subclauses on objects and named numbers, to cut down on the number of forward references. 13.a

13.b We have dropped the term "base type" in favor of simply "type" (all types in Ada 83 were "base types" so it wasn't clear when it was appropriate/necessary to say "base type"). Given a subtype S of a type T, we call T the "type of the subtype S."

Wording Changes from Ada 95

13.c/2 {*AI95-00231-01*} Added a mention of null exclusions when we're talking about constraints (these are not constraints, but they are similar).

13.d/2 {*AI95-00251-01*} Defined an interface type to be a composite type.

13.e/2 {*AI95-00326-01*} Revised the wording so that it is clear that an incomplete view is similar to a partial view in terms of the language.

13.f/2 {*AI95-00366-01*} Added a definition of component of a type, subcomponent of a type, and part of a type. These are commonly used in the standard, but they were not previously defined.

13.g/2 {*AI95-00442-01*} Reworded most of this clause to use category rather than class, since so many interesting properties are not, strictly speaking, classes. Moreover, there was no normative description of exactly which properties formed classes, and which did not. The real definition of class, along with a list of properties, is now in 3.4.

3.2.1 Type Declarations

1 A type_declaration declares a type and its first subtype.

Syntax

```
2    type_declaration ::=  full_type_declaration
        | incomplete_type_declaration
        | private_type_declaration
        | private_extension_declaration

3    full_type_declaration ::=
          type defining_identifier [known_discriminant_part] is type_definition;
        | task_type_declaration
        | protected_type_declaration

4/2  {AI95-00251-01} type_definition ::=
          enumeration_type_definition   | integer_type_definition
        | real_type_definition          | array_type_definition
        | record_type_definition        | access_type_definition
        | derived_type_definition       | interface_type_definition
```

Legality Rules

5 A given type shall not have a subcomponent whose type is the given type itself.

Static Semantics

6 {*first subtype*} The defining_identifier of a type_declaration denotes the *first subtype* of the type. The known_discriminant_part, if any, defines the discriminants of the type (see 3.7, “Discriminants”). The remainder of the type_declaration defines the remaining characteristics of (the view of) the type.

7/2 {*AI95-00230-01*} {*named type*} A type defined by a type_declaration is a *named* type; such a type has one or more nameable subtypes. {*anonymous type*} Certain other forms of declaration also include type definitions as part of the declaration for an object. The type defined by such a declaration is *anonymous* — it has no nameable subtypes. {*italics (pseudo-names of anonymous types)*} For explanatory purposes, this International Standard sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier. For a named type whose first subtype is T, this International Standard sometimes refers to the type of T as simply “the type T”.

Ramification: {*AI95-00230-01*} The only user-defined types that can be anonymous in the above sense are array, access, task, and protected types. An anonymous array, task, or protected type can be defined as part of an object_declaration. An anonymous access type can be defined as part of numerous other constructs. 7.a/2

{*AI95-00230-01*} {*AI95-00326-01*} {*full type*} A named type that is declared by a full_type_declaration, or an anonymous type that is defined by an access_definition or as part of declaring an object of the type, is called a *full type*.{*full type definition*} The declaration of a full type also declares the *full view* of the type.{*full view (of a type)*} The type_definition, task_definition, protected_definition, or access_-definition that defines a full type is called a *full type definition*. [Types declared by other forms of type_-declaration are not separate types; they are partial or incomplete views of some full type.] 8/2

To be honest: Class-wide, universal, and root numeric types are full types. 8.a

Reason: {*AI95-00230-01*} We need to mention access_definition separately, as it may occur in renames, which do not declare objects. 8.b/2

{*predefined operator* [partial]} The definition of a type implicitly declares certain *predefined operators* that operate on the type, according to what classes the type belongs, as specified in 4.5, “Operators and Expression Evaluation”. 9

Discussion: We no longer talk about the implicit declaration of basic operations. These are treated like an if_statement — they don't need to be declared, but are still applicable to only certain classes of types. 9.a

{*predefined type*} The *predefined types* [(for example the types Boolean, Wide_Character, Integer, *root_integer*, and *universal_integer*)] are the types that are defined in [a predefined library package called] Standard[; this package also includes the [(implicit)] declarations of their predefined operators]. [The package Standard is described in A.1.] 10

Ramification: We use the term “predefined” to refer to entities declared in the visible part of Standard, to implicitly declared operators of a type whose semantics are defined by the language, to Standard itself, and to the “predefined environment”. We do not use this term to refer to library packages other than Standard. For example Text_IO is a language-defined package, not a predefined package, and Text_IO.Put_Line is not a predefined operation. 10.a

Dynamic Semantics

{*elaboration (full_type_declaration)* [partial]} The elaboration of a full_type_declaration consists of the elaboration of the full type definition. {*elaboration (full type definition)* [partial]} Each elaboration of a full type definition creates a distinct type and its first subtype. 11

Reason: The creation is associated with the type *definition*, rather than the type *declaration*, because there are types that are created by full type definitions that are not immediately contained within a type declaration (e.g. an array object declaration, a singleton task declaration, etc.). 11.a

Ramification: Any implicit declarations that occur immediately following the full type definition are elaborated where they (implicitly) occur. 11.b

Examples

Examples of type definitions: 12

13
```
(White, Red, Yellow, Green, Blue, Brown, Black)
range 1 .. 72
array(1 .. 10) of Integer
```

Examples of type declarations: 14

15
```
type Color  is (White, Red, Yellow, Green, Blue, Brown, Black);
type Column is range 1 .. 72;
type Table  is array(1 .. 10) of Integer;
```

NOTES

3 Each of the above examples declares a named type. The identifier given denotes the first subtype of the type. Other named subtypes of the type can be declared with subtype_declarations (see 3.2.2). Although names do not directly denote types, a phrase like “the type Column” is sometimes used in this International Standard to refer to the type of Column, where Column denotes the first subtype of the type. For an example of the definition of an anonymous type, see the declaration of the array Color_Table in 3.3.1; its type is anonymous — it has no nameable subtypes. 16

Wording Changes from Ada 83

16.a The syntactic category full_type_declaration now includes task and protected type declarations.

16.b We have generalized the concept of first-named subtype (now called simply “first subtype”) to cover all kinds of types, for uniformity of description elsewhere. RM83 defined first-named subtype in Section 13. We define first subtype here, because it is now a more fundamental concept. We renamed the term, because in Ada 95 some first subtypes have no name.

16.c/2 {*AI95-00230-01*} We no longer elaborate discriminant_parts, because there is nothing to do, and it was complex to say that you only wanted to elaborate it once for a private or incomplete type. This is also consistent with the fact that subprogram specifications are not elaborated (neither in Ada 83 nor in Ada 95). Note, however, that an access_definition appearing in a discriminant_part is elaborated at the full_type_declaration (for a nonlimited type) or when an object with such a discriminant is created (for a limited type).

Wording Changes from Ada 95

16.d/2 {*AI95-00230-01*} Added wording so that anonymous access types are always full types, even if they appear in renames.

16.e/2 {*AI95-00251-01*} Added interface types (see 3.9.4) to the syntax.

16.f/2 {*AI95-00326-01*} Added a definition of full view, so that all types have a well-defined full view.

3.2.2 Subtype Declarations

1 A subtype_declaration declares a subtype of some previously declared type, as defined by a subtype_indication.

Syntax

2 subtype_declaration ::=
  subtype defining_identifier **is** subtype_indication;

3/2 {*AI95-00231-01*} subtype_indication ::= [null_exclusion] subtype_mark [constraint]

4 subtype_mark ::= *subtype*_name

4.a **Ramification:** Note that name includes attribute_reference; thus, S'Base can be used as a subtype_mark.

4.b **Reason:** We considered changing subtype_mark to subtype_name. However, existing users are used to the word "mark," so we're keeping it.

5 constraint ::= scalar_constraint | composite_constraint

6 scalar_constraint ::=
  range_constraint | digits_constraint | delta_constraint

7 composite_constraint ::=
  index_constraint | discriminant_constraint

Name Resolution Rules

8 A subtype_mark shall resolve to denote a subtype. {*determines (a type by a subtype_mark)*} The type *determined by* a subtype_mark is the type of the subtype denoted by the subtype_mark.

8.a **Ramification:** Types are never directly named; all subtype_marks denote subtypes — possibly an unconstrained (base) subtype, but never the type. When we use the term *anonymous type* we really mean a type with no namable subtypes.

Dynamic Semantics

9 {*elaboration (subtype_declaration)* [partial]} The elaboration of a subtype_declaration consists of the elaboration of the subtype_indication. {*elaboration (subtype_indication)* [partial]} The elaboration of a subtype_indication creates a new subtype. If the subtype_indication does not include a constraint, the new subtype has the same (possibly null) constraint as that denoted by the subtype_mark. The elaboration of a subtype_indication that includes a constraint proceeds as follows:

- The constraint is first elaborated. 10
- {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} A check is then made that the constraint is *compatible* with the subtype denoted by the subtype_mark. 11

Ramification: The checks associated with constraint compatibility are all Range_Checks. Discriminant_Checks and Index_Checks are associated only with checks that a value satisfies a constraint. 11.a

The condition imposed by a constraint is the condition obtained after elaboration of the constraint. {*compatibility (constraint with a subtype)* [distributed]} The rules defining compatibility are given for each form of constraint in the appropriate subclause. These rules are such that if a constraint is *compatible* with a subtype, then the condition imposed by the constraint cannot contradict any condition already imposed by the subtype on its values. {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised if any check of compatibility fails. 12

To be honest: The condition imposed by a constraint is named after it — a range_constraint imposes a range constraint, etc. 12.a

Ramification: A range_constraint causes freezing of its type. Other constraints do not. 12.b

NOTES

4 A scalar_constraint may be applied to a subtype of an appropriate scalar type (see 3.5, 3.5.9, and J.3), even if the subtype is already constrained. On the other hand, a composite_constraint may be applied to a composite subtype (or an access-to-composite subtype) only if the composite subtype is unconstrained (see 3.6.1 and 3.7.1). 13

Examples

Examples of subtype declarations: 14

15/2

```
{AI95-00433-01} subtype Rainbow   is Color range Red .. Blue;        -- see 3.2.1
subtype Red_Blue  is Rainbow;
subtype Int       is Integer;
subtype Small_Int is Integer range -10 .. 10;
subtype Up_To_K   is Column range 1 .. K;            -- see 3.2.1
subtype Square    is Matrix(1 .. 10, 1 .. 10);       -- see 3.6
subtype Male      is Person(Sex => M);               -- see 3.10.1
subtype Binop_Ref is not null Binop_Ptr;             -- see 3.10
```

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} In Ada 95, all range_constraints cause freezing of their type. Hence, a type-related representation item for a scalar type has to precede any range_constraints whose type is the scalar type. 15.a

Wording Changes from Ada 83

Subtype_marks allow only subtype names now, since types are never directly named. There is no need for RM83-3.3.2(3), which says a subtype_mark can denote both the type and the subtype; in Ada 95, you denote an unconstrained (base) subtype if you want, but never the type. 15.b

The syntactic category type_mark is now called subtype_mark, since it always denotes a subtype. 15.c

Extensions to Ada 95

{*AI95-00231-01*} {*extensions to Ada 95*} An optional null_exclusion can be used in a subtype_indication. This is described in 3.10 15.d/2

3.2.3 Classification of Operations

Static Semantics

{*AI95-00416-01*} {*operates on a type*} An operation *operates on a type T* if it yields a value of type *T*, if it has an operand whose expected type (see 8.6) is *T*, or if it has an access parameter or access result type (see 6.1) designating *T*. {*predefined operation (of a type)*} A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined* 1/2 *operation* of the type. {*primitive operations (of a type)*} The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms.

1.a **Glossary entry:** {*Primitive operations*} The primitive operations of a type are the operations (such as subprograms) declared together with the type declaration. They are inherited by other types in the same class of types. For a tagged type, the primitive subprograms are dispatching subprograms, providing run-time polymorphism. A dispatching subprogram may be called with statically tagged operands, in which case the subprogram body invoked is determined at compile time. Alternatively, a dispatching subprogram may be called using a dispatching call, in which case the subprogram body invoked is determined at run time.

1.b **To be honest:** Protected subprograms are not considered to be "primitive subprograms," even though they are subprograms, and they are inherited by derived types.

1.c **Discussion:** We use the term "primitive subprogram" in most of the rest of the manual. The term "primitive operation" is used mostly in conceptual discussions.

2 {*primitive subprograms (of a type)*} The *primitive subprograms* of a specific type are defined as follows:

3 • The predefined operators of the type (see 4.5);

4 • For a derived type, the inherited (see 3.4) user-defined subprograms;

5 • For an enumeration type, the enumeration literals (which are considered parameterless functions — see 3.5.1);

6 • For a specific type declared immediately within a package_specification, any subprograms (in addition to the enumeration literals) that are explicitly declared immediately within the same package_specification and that operate on the type;

7/2 • {*AI95-00200-01*} {*override (a primitive subprogram)*} For a nonformal type, any subprograms not covered above [that are explicitly declared immediately within the same declarative region as the type] and that override (see 8.3) other implicitly declared primitive subprograms of the type.

7.a **Discussion:** In Ada 83, only subprograms declared in the visible part were "primitive" (i.e. derivable). In Ada 95, mostly because of child library units, we include all operations declared in the private part as well, and all operations that override implicit declarations.

7.b **Ramification:** It is possible for a subprogram to be primitive for more than one type, though it is illegal for a subprogram to be primitive for more than one tagged type. See 3.9.

7.c **Discussion:** The order of the implicit declarations when there are both predefined operators and inherited subprograms is described in 3.4, "Derived Types and Classes".

7.d/2 **Ramification:** {*AI95-00200-01*} Subprograms declared in a generic package specification are never primitive for a formal type, even if they happen to override an operation of the formal type. This includes formal subprograms, which are never primitive operations (that's true even for an abstract formal subprogram).

8 {*primitive operator (of a type)*} A primitive subprogram whose designator is an operator_symbol is called a *primitive operator*.

Incompatibilities With Ada 83

8.a {*incompatibilities with Ada 83*} The attribute S'Base is no longer defined for non-scalar subtypes. Since this was only permitted as the prefix of another attribute, and there are no interesting non-scalar attributes defined for an unconstrained composite or access subtype, this should not affect any existing programs.

Extensions to Ada 83

8.b {*extensions to Ada 83*} The primitive subprograms (derivable subprograms) include subprograms declared in the private part of a package specification as well, and those that override implicitly declared subprograms, even if declared in a body.

Wording Changes from Ada 83

8.c We have dropped the confusing term *operation of a type* in favor of the more useful *primitive operation of a type* and the phrase *operates on a type*.

8.d The description of S'Base has been moved to 3.5, "Scalar Types" because it is now defined only for scalar types.

Wording Changes from Ada 95

{*AI95-00200-01*} Clarified that a formal subprogram that happens to override a primitive operation of a formal type is not a primitive operation (and thus not a dispatching operation) of the formal type. 8.e/2

{*AI95-00416-01*} Added wording to include access result types in the kinds of operations that operate on a type T. 8.f/2

3.3 Objects and Named Numbers

1 [Objects are created at run time and contain a value of a given type. {*creation (of an object)*} An object can be created and initialized as part of elaborating a declaration, evaluating an allocator, aggregate, or function_call, or passing a parameter by copy. Prior to reclaiming the storage for an object, it is finalized if necessary (see 7.6.1).]

Static Semantics

2 {*object*} All of the following are objects:

2.a **Glossary entry:** {*Object*} An object is either a constant or a variable. An object contains a value. An object is created by an object_declaration or by an allocator. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

- 3 the entity declared by an object_declaration;
- 4 a formal parameter of a subprogram, entry, or generic subprogram;
- 5 a generic formal object;
- 6 a loop parameter;
- 7 a choice parameter of an exception_handler;
- 8 an entry index of an entry_body;
- 9 the result of dereferencing an access-to-object value (see 4.1);
- 10/2 {*AI95-00416-01*} the return object created as the result of evaluating a function_call (or the equivalent operator invocation — see 6.6);
- 11 the result of evaluating an aggregate;
- 12 a component, slice, or view conversion of another object.

13 {*constant*} {*variable*} {*constant object*} {*variable object*} {*constant view*} {*variable view*} An object is either a *constant* object or a *variable* object. The value of a constant object cannot be changed between its initialization and its finalization, whereas the value of a variable object can be changed. Similarly, a view of an object is either a *constant* or a *variable*. All views of a constant object are constant. A constant view of a variable object cannot be used to modify the value of the variable. The terms constant and variable by themselves refer to constant and variable views of objects.

14 {*read (the value of an object)*} The value of an object is *read* when the value of any part of the object is evaluated, or when the value of an enclosing object is evaluated. {*update (the value of an object)*} The value of a variable is *updated* when an assignment is performed to any part of the variable, or when an assignment is performed to an enclosing object.

14.a **Ramification:** Reading and updating are intended to include read/write references of any kind, even if they are not associated with the evaluation of a particular construct. Consider, for example, the expression “X.**all**(F)”, where X is an access-to-array object, and F is a function. The implementation is allowed to first evaluate “X.**all**” and then F. Finally, a read is performed to get the value of the F'th component of the array. Note that the array is not necessarily read as part of the evaluation of “X.**all**”. This is important, because if F were to free X using Unchecked_Deallocation, we want the execution of the final read to be erroneous.

15 Whether a view of an object is constant or variable is determined by the definition of the view. The following (and no others) represent constants:

16 • an object declared by an object_declaration with the reserved word **constant**;

16.a/2 **To be honest:** {*AI95-00385-01*} We mean the word **constant** as defined by the grammar for object_declaration, not some random word **constant**. Thus,

16.b/2
```
X : access constant T;
```

16.c/2 is not a constant.

17 • a formal parameter or generic formal object of mode **in**;

18 • a discriminant;

19 • a loop parameter, choice parameter, or entry index;

20 • the dereference of an access-to-constant value;

21 • the result of evaluating a function_call or an aggregate;

22 • a selected_component, indexed_component, slice, or view conversion of a constant.

23 {*nominal subtype*} At the place where a view of an object is defined, a *nominal subtype* is associated with the view. {*actual subtype*} {*subtype (of an object): See actual subtype of an object*} The object's *actual subtype* (that is, its subtype) can be more restrictive than the nominal subtype of the view; it always is if the nominal subtype is an *indefinite subtype*. {*indefinite subtype*} {*definite subtype*} A subtype is an indefinite subtype if it is an unconstrained array subtype, or if it has unknown discriminants or unconstrained discriminants without defaults (see 3.7); otherwise the subtype is a *definite* subtype [(all elementary subtypes are definite subtypes)]. [A class-wide subtype is defined to have unknown discriminants, and is therefore an indefinite subtype. An indefinite subtype does not by itself provide enough information to create an object; an additional constraint or explicit initialization expression is necessary (see 3.3.1). A component cannot have an indefinite nominal subtype.]

24 {*named number*} A *named number* provides a name for a numeric value known at compile time. It is declared by a number_declaration.

NOTES

25 5 A constant cannot be the target of an assignment operation, nor be passed as an **in out** or **out** parameter, between its initialization and finalization, if any.

26 6 The nominal and actual subtypes of an elementary object are always the same. For a discriminated or array object, if the nominal subtype is constrained then so is the actual subtype.

Extensions to Ada 83

26.a {*extensions to Ada 83*} There are additional kinds of objects (choice parameters and entry indices of entry bodies).

26.b The result of a function and of evaluating an aggregate are considered (constant) objects. This is necessary to explain the action of finalization on such things. Because a function_call is also syntactically a name (see 4.1), the result of a function_call can be renamed, thereby allowing repeated use of the result without calling the function again.

Wording Changes from Ada 83

26.c This clause and its subclauses now follow the clause and subclauses on types and subtypes, to cut down on the number of forward references.

26.d The term nominal subtype is new. It is used to distinguish what is known at compile time about an object's constraint, versus what its "true" run-time constraint is.

26.e The terms definite and indefinite (which apply to subtypes) are new. They are used to aid in the description of generic formal type matching, and to specify when an explicit initial value is required in an object_declaration.

26.f We have moved the syntax for object_declaration and number_declaration down into their respective subclauses, to keep the syntax close to the description of the associated semantics.

26.g We talk about variables and constants here, since the discussion is not specific to object_declarations, and it seems better to have the list of the kinds of constants juxtaposed with the kinds of objects.

26.h We no longer talk about indirect updating due to parameter passing. Parameter passing is handled in 6.2 and 6.4.1 in a way that there is no need to mention it here in the definition of read and update. Reading and updating now includes the case of evaluating or assigning to an enclosing object.

Wording Changes from Ada 95

{*AI95-00416-01*} Clarified that the return object is the object created by a function call. 26.i/2

3.3.1 Object Declarations

1 {*stand-alone object* [distributed]} {*explicit initial value*} {*initialization expression*} An object_declaration declares a *stand-alone* object with a given nominal subtype and, optionally, an explicit initial value given by an initialization expression. {*anonymous array type*} {*anonymous task type*} {*anonymous protected type*} For an array, task, or protected object, the object_declaration may include the definition of the (anonymous) type of the object.

Syntax

2/2 {*AI95-00385-01*} {*AI95-00406-01*} object_declaration ::=
  defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression];
 | defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression];
 | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression];
 | single_task_declaration
 | single_protected_declaration

3 defining_identifier_list ::=
 defining_identifier {, defining_identifier}

Name Resolution Rules

4 {*expected type (object_declaration initialization expression)* [partial]} For an object_declaration with an expression following the compound delimiter :=, the type expected for the expression is that of the object. {*initialization expression*} This expression is called the *initialization expression*. {*constructor: See initialization expression*}

Legality Rules

5/2 {*AI95-00287-01*} An object_declaration without the reserved word **constant** declares a variable object. If it has a subtype_indication or an array_type_definition that defines an indefinite subtype, then there shall be an initialization expression.

Static Semantics

6 An object_declaration with the reserved word **constant** declares a constant object. {*full constant declaration*} If it has an initialization expression, then it is called a *full constant declaration*. {*deferred constant declaration*} Otherwise it is called a *deferred constant declaration*. The rules for deferred constant declarations are given in clause 7.4. The rules for full constant declarations are given in this subclause.

7 Any declaration that includes a defining_identifier_list with more than one defining_identifier is equivalent to a series of declarations each containing one defining_identifier from the list, with the rest of the text of the declaration copied for each declaration in the series, in the same order as the list. The remainder of this International Standard relies on this equivalence; explanations are given for declarations with a single defining_identifier.

8/2 {*AI95-00385-01*} {*nominal subtype*} The subtype_indication, access_definition, or full type definition of an object_declaration defines the nominal subtype of the object. The object_declaration declares an object of the type of the nominal subtype.

8.a/2 **Discussion:** {*AI95-00385-01*} The phrase “full type definition” here includes the case of an anonymous array, access, task, or protected type.

8.1/2 {*AI95-00373-01*} {*requires late initialization*} A component of an object is said to *require late initialization* if it has an access discriminant value constrained by a per-object expression, or if it has an initialization expression that includes a name denoting the current instance of the type or denoting an access discriminant.

8.b/2 **Reason:** Such components can depend on the values of other components of the object. We want to initialize them as late and as reproducibly as possible.

Dynamic Semantics

9/2 {*AI95-00363-01*} {*constraint (of an object)*} If a composite object declared by an object_declaration has an unconstrained nominal subtype, then if this subtype is indefinite or the object is constant the actual subtype of this object is constrained. The constraint is determined by the bounds or discriminants (if any) of its initial value; {*constrained by its initial value*} the object is said to be *constrained by its initial value*. {*actual subtype (of an object)*} {*subtype (of an object): See actual subtype of an object*} When not constrained by its initial value, the actual and nominal subtypes of the object are the same. {*constrained (object)*} {*unconstrained (object)*} If its actual subtype is constrained, the object is called a *constrained object*.

10 {*implicit initial values (for a subtype)*} For an object_declaration without an initialization expression, any initial values for the object or its subcomponents are determined by the *implicit initial values* defined for its nominal subtype, as follows:

11 • The implicit initial value for an access subtype is the null value of the access type.

12 • The implicit initial (and only) value for each discriminant of a constrained discriminated subtype is defined by the subtype.

13 • For a (definite) composite subtype, the implicit initial value of each component with a default_expression is obtained by evaluation of this expression and conversion to the component's nominal subtype (which might raise Constraint_Error — see 4.6, “Type Conversions”), unless the component is a discriminant of a constrained subtype (the previous case), or is in an excluded variant (see 3.8.1). {*implicit subtype conversion (component defaults)* [partial]} For each component that does not have a default_expression, any implicit initial values are those determined by the component's nominal subtype.

14 • For a protected or task subtype, there is an implicit component (an entry queue) corresponding to each entry, with its implicit initial value being an empty queue.

14.a **Implementation Note:** The implementation may add implicit components for its own use, which might have implicit initial values. For a task subtype, such components might represent the state of the associated thread of control. For a type with dynamic-sized components, such implicit components might be used to hold the offset to some explicit component.

15 {*elaboration (object_declaration)* [partial]} The elaboration of an object_declaration proceeds in the following sequence of steps:

16/2 1. {*AI95-00385-01*} The subtype_indication, access_definition, array_type_definition, single_-task_declaration, or single_protected_declaration is first elaborated. This creates the nominal subtype (and the anonymous type in the last four cases).

17 2. If the object_declaration includes an initialization expression, the (explicit) initial value is obtained by evaluating the expression and converting it to the nominal subtype (which might raise Constraint_Error — see 4.6). {*implicit subtype conversion (initialization expression)* [partial]}

18/2 3. {*8652/0002*} {*AI95-00171-01*} {*AI95-00373-01*} The object is created, and, if there is not an initialization expression, the object is *initialized by default*. {*initialized by default*} When an object is initialized by default, any per-object constraints (see 3.8) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype. {*initialization (of an object)*} {*assignment operation (during elaboration of an object_declaration)*} Any initial values (whether explicit or implicit) are assigned to the object or to the corresponding subcomponents. As described in 5.2 and 7.6, Initialize and Adjust procedures can be called. {*constructor: See initialization*}

Discussion: For a per-object constraint that contains some per-object expressions and some non-per-object expressions, the values used for the constraint consist of the values of the non-per-object expressions evaluated at the point of the type_declaration, and the values of the per-object expressions evaluated at the point of the creation of the object. 18.a

The elaboration of per-object constraints was presumably performed as part of the dependent compatibility check in Ada 83. If the object is of a limited type with an access discriminant, the access_definition is elaborated at this time (see 3.7). 18.b

Reason: The reason we say that evaluating an explicit initialization expression happens before creating the object is that in some cases it is impossible to know the size of the object being created until its initial value is known, as in “X: String := Func_Call(...);”. The implementation can create the object early in the common case where the size can be known early, since this optimization is semantically neutral. 18.c

This paragraph was deleted.{*AI95-00373-01*} 19/2

Ramification: Since the initial values have already been converted to the appropriate nominal subtype, the only Constraint_Errors that might occur as part of these assignments are for values outside their base range that are used to initialize unconstrained numeric subcomponents. See 3.5. 19.a

{*AI95-00373-01*} For the third step above, evaluations and assignments are performed in an arbitrary order subject to the following restrictions: 20/2

- {*AI95-00373-01*} Assignment to any part of the object is preceded by the evaluation of the value that is to be assigned. 20.1/2

Reason: Duh. But we ought to say it. Note that, like any rule in the International Standard, it doesn't prevent an “as-if” optimization; as long as the semantics as observed from the program are correct, the compiler can generate any code it wants. 20.a.1/2

- {*AI95-00373-01*} The evaluation of a default_expression that includes the name of a discriminant is preceded by the assignment to that discriminant. 20.2/2

Reason: Duh again. But we have to say this, too. It's odd that Ada 95 only required the default expressions to be evaluated before the discriminant is used; it says nothing about discriminant values that come from subtype_indications. 20.a.2/2

- {*AI95-00373-01*} The evaluation of the default_expression for any component that depends on a discriminant is preceded by the assignment to that discriminant. 20.3/2

Reason: For example: 20.a

20.b
```
type R(D : Integer := F) is
    record
        S : String(1..D) := (others => G);
    end record;
```

20.c
```
X : R;
```

For the elaboration of the declaration of X, it is important that F be evaluated before the aggregate. 20.d

- {*AI95-00373-01*} The assignments to any components, including implicit components, not requiring late initialization must precede the initial value evaluations for any components requiring late initialization; if two components both require late initialization, then assignments to parts of the component occurring earlier in the order of the component declarations must precede the initial value evaluations of the component occurring later. 20.4/2

Reason: Components that require late initialization can refer to the entire object during their initialization. We want them to be initialized as late as possible to reduce the chance that their initialization depends on uninitialized components. For instance: 20.e/2

20.f/2
```
type T (D : Natural) is
  limited record
    C1 : T1 (T'Access);
    C2 : Natural := F (D);
    C3 : String (1 .. D) := (others => ' ');
  end record;
```

20.g/2 Component C1 requires late initialization. The initialization could depend on the values of any component of T, including D, C2, or C3. Therefore, we want to it to be initialized last. Note that C2 and C3 do not require late initialization; they only have to be initialized after D.

20.h/2 It is possible for there to be more than one component that requires late initialization. In this case, the language can't prevent problems, because all of the components can't be the last one initialized. In this case, we specify the order of initialization for components requiring late initialization; by doing so, programmers can arrange their code to avoid accessing uninitialized components, and such arrangements are portable. Note that if the program accesses an uninitialized component, 13.9.1 defines the execution to be erroneous.

21 [There is no implicit initial value defined for a scalar subtype.] {*uninitialized variables* [partial]} In the absence of an explicit initialization, a newly created scalar object might have a value that does not belong to its subtype (see 13.9.1 and H.1).

21.a **To be honest:** It could even be represented by a bit pattern that doesn't actually represent any value of the type at all, such as an invalid internal code for an enumeration type, or a NaN for a floating point type. It is a generally a bounded error to reference scalar objects with such “invalid representations”, as explained in 13.9.1, “Data Validity”.

21.b **Ramification:** There is no requirement that two objects of the same scalar subtype have the same implicit initial “value” (or representation). It might even be the case that two elaborations of the same object_declaration produce two different initial values. However, any particular uninitialized object is default-initialized to a single value (or invalid representation). Thus, multiple reads of such an uninitialized object will produce the same value each time (if the implementation chooses not to detect the error).

NOTES

22 7 Implicit initial values are not defined for an indefinite subtype, because if an object's nominal subtype is indefinite, an explicit initial value is required.

23 8 {*stand-alone constant*} {*stand-alone variable*} As indicated above, a stand-alone object is an object declared by an object_declaration. Similar definitions apply to “stand-alone constant” and “stand-alone variable.” A subcomponent of an object is not a stand-alone object, nor is an object that is created by an allocator. An object declared by a loop_parameter_specification, parameter_specification, entry_index_specification, choice_parameter_specification, or a formal_object_declaration is not called a stand-alone object.

24 9 The type of a stand-alone object cannot be abstract (see 3.9.3).

Examples

25 *Example of a multiple object declaration:*

26
```
-- the multiple object declaration
```

27/2
```
{AI95-00433-01} John, Paul : not null Person_Name := new Person(Sex => M);  -- see 3.10.1
```

28
```
-- is equivalent to the two single object declarations in the order given
```

29/2
```
{AI95-00433-01} John : not null Person_Name := new Person(Sex => M);
Paul : not null Person_Name := new Person(Sex => M);
```

30 *Examples of variable declarations:*

31/2
```
{AI95-00433-01} Count, Sum  : Integer;
Size        : Integer range 0 .. 10_000 := 0;
Sorted      : Boolean := False;
Color_Table : array(1 .. Max) of Color;
Option      : Bit_Vector(1 .. 10) := (others => True);
Hello       : aliased String := "Hi, world.";
θ, φ        : Float range -π .. +π;
```

32 *Examples of constant declarations:*

33/2
```
{AI95-00433-01} Limit     : constant Integer := 10_000;
Low_Limit : constant Integer := Limit/10;
Tolerance : constant Real := Dispersion(1.15);
Hello_Msg : constant access String := Hello'Access; -- see 3.10.2
```

Extensions to Ada 83

33.a {*extensions to Ada 83*} The syntax rule for object_declaration is modified to allow the **aliased** reserved word.

A variable declared by an object_declaration can be constrained by its initial value; that is, a variable of a nominally unconstrained array subtype, or discriminated type without defaults, can be declared so long as it has an explicit initial value. In Ada 83, this was permitted for constants, and for variables created by allocators, but not for variables declared by object_declarations. This is particularly important for tagged class-wide types, since there is no way to constrain them explicitly, and so an initial value is the only way to provide a constraint. It is also important for generic formal private types with unknown discriminants. 33.b

We now allow an unconstrained_array_definition in an object_declaration. This allows an object of an anonymous array type to have its bounds determined by its initial value. This is for uniformity: If one can write "X: **constant array**(Integer **range** 1..10) **of** Integer := ...;" then it makes sense to also allow "X: **constant array**(Integer **range** <>) **of** Integer := ...;". (Note that if anonymous array types are ever sensible, a common situation is for a table implemented as an array. Tables are often constant, and for constants, there's usually no point in forcing the user to count the number of elements in the value.) 33.c

Wording Changes from Ada 83

We have moved the syntax for object_declarations into this subclause. 33.d

Deferred constants no longer have a separate syntax rule, but rather are incorporated in object_declaration as constants declared without an initialization expression. 33.e

Inconsistencies With Ada 95

{*AI95-00363-01*} {*inconsistencies with Ada 95*} Unconstrained aliased objects of types with discriminants with defaults are no longer constrained by their initial values. This means that a program that raised Constraint_Error from an attempt to change the discriminants will no longer do so. The change only affects programs that depended on the raising of Constraint_Error in this case, so the inconsistency is unlikely to occur outside of the ACATS. This change may however cause compilers to implement these objects differently, possibly taking additional memory or time. This is unlikely to be worse than the differences caused by any major compiler upgrade. 33.f/2

Extensions to Ada 95

{*AI95-00287-01*} {*extensions to Ada 95*} A constant may have a limited type; the initialization expression has to be built-in-place (see 7.5). 33.g/2

{*AI95-00385-01*} {*AI95-00406-01*} {*extensions to Ada 95*} A stand-alone object may have an anonymous access type. 33.h/2

Wording Changes from Ada 95

{*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Corrected wording to say that per-object constraints are elaborated (not evaluated). 33.i/2

{*AI95-00373-01*} The rules for evaluating default initialization have been tightened. In particular, components whose default initialization can refer to the rest of the object are required to be initialized last. 33.j/2

{*AI95-00433-01*} Added examples of various new constructs. 33.k/2

3.3.2 Number Declarations

A number_declaration declares a named number. 1

Discussion: {*static*} If a value or other property of a construct is required to be *static* that means it is required to be determined prior to execution. A *static* expression is an expression whose value is computed at compile time and is usable in contexts where the actual value might affect the legality of the construct. This is fully defined in clause 4.9. 1.a

Syntax

number_declaration ::= 2
    defining_identifier_list : **constant** := *static*_expression;

Name Resolution Rules

{*expected type (number_declaration expression)* [partial]} The *static*_expression given for a number_declaration is expected to be of any numeric type. 3

Legality Rules

4 The *static*_expression given for a number declaration shall be a static expression, as defined by clause 4.9.

Static Semantics

5 The named number denotes a value of type *universal_integer* if the type of the *static*_expression is an integer type. The named number denotes a value of type *universal_real* if the type of the *static*_-expression is a real type.

6 The value denoted by the named number is the value of the *static*_expression, converted to the corresponding universal type. {*implicit subtype conversion (named number value)* [partial]}

Dynamic Semantics

7 {*elaboration (number_declaration)* [partial]} The elaboration of a number_declaration has no effect.

7.a **Proof:** Since the *static*_expression was evaluated at compile time.

Examples

8 *Examples of number declarations:*

9
```
Two_Pi        : constant := 2.0*Ada.Numerics.Pi;   -- a real number (see A.5)
```

10/2
```
{AI95-00433-01} Max           : constant := 500;                   -- an integer number
Max_Line_Size : constant := Max/6;                 -- the integer 83
Power_16      : constant := 2**16;                 -- the integer 65_536
One, Un, Eins : constant := 1;                     -- three different names for 1
```

Extensions to Ada 83

10.a {*extensions to Ada 83*} We now allow a static expression of any numeric type to initialize a named number. For integer types, it was possible in Ada 83 to use 'Pos to define a named number, but there was no way to use a static expression of some non-universal real type to define a named number. This change is upward compatible because of the preference rule for the operators of the root numeric types.

Wording Changes from Ada 83

10.b We have moved the syntax rule into this subclause.

10.c AI83-00263 describes the elaboration of a number declaration in words similar to that of an object_declaration. However, since there is no expression to be evaluated and no object to be created, it seems simpler to say that the elaboration has no effect.

3.4 Derived Types and Classes

1/2 {*AI95-00401-01*} {*AI95-00419-01*} {*derived type*} A derived_type_definition defines a *derived type* (and its first subtype) whose characteristics are *derived* from those of a parent type, and possibly from progenitor types. {*inheritance: See derived types and classes*}

1.a/2 **Glossary entry:** {*Derived type*} A derived type is a type defined in terms of one or more other types given in a derived type definition. The first of those types is the parent type of the derived type and any others are progenitor types. Each class containing the parent type or a progenitor type also contains the derived type. The derived type inherits properties such as components and primitive operations from the parent and progenitors. A type together with the types derived from it (directly or indirectly) form a derivation class.

1.1/2 {*AI95-00442-01*} {*class (of types)*} {*category (of types)*} A *class of types* is a set of types that is closed under derivation; that is, if the parent or a progenitor type of a derived type belongs to a class, then so does the derived type. By saying that a particular group of types forms a class, we are saying that all derivatives of a type in the set inherit the characteristics that define that set. The more general term *category of types* is used for a set of types whose defining characteristics are not necessarily inherited by derivatives; for example, limited, abstract, and interface are all categories of types, but not classes of types.

Ramification: A class of types is also a category of types. 1.b/2

Syntax

{*AI95-00251-01*} {*AI95-00419-01*} derived_type_definition ::= 2/2
  [**abstract**] [**limited**] **new** *parent_*subtype_indication [[**and** interface_list] record_extension_part]

Legality Rules

{*AI95-00251-01*} {*AI95-00401-01*} {*AI95-00419-01*} {*parent subtype*} {*parent type*} The *parent_*subtype_indication defines the *parent subtype*; its type is the *parent type*. The interface_list defines the progenitor types (see 3.9.4). A derived type has one parent type and zero or more progenitor types. 3/2

Glossary entry: {*Parent*} The parent of a derived type is the first type given in the definition of the derived type. The parent can be almost any kind of type, including an interface type. 3.a/2

A type shall be completely defined (see 3.11.1) prior to being specified as the parent type in a derived_type_definition — [the full_type_declarations for the parent type and any of its subcomponents have to precede the derived_type_definition.] 4

Discussion: This restriction does not apply to the ancestor type of a private extension — see 7.3; such a type need not be completely defined prior to the private_extension_declaration. However, the restriction does apply to record extensions, so the ancestor type will have to be completely defined prior to the full_type_declaration corresponding to the private_extension_declaration. 4.a

Reason: We originally hoped we could relax this restriction. However, we found it too complex to specify the rules for a type derived from an incompletely defined limited type that subsequently became nonlimited. 4.b

{*AI95-00401-01*} {*record extension*} If there is a record_extension_part, the derived type is called a *record extension* of the parent type. A record_extension_part shall be provided if and only if the parent type is a tagged type. [An interface_list shall be provided only if the parent type is a tagged type.] 5/2

Proof: {*AI95-00401-01*} The syntax only allows an interface_list to appear with a record_extension_part, and a record_extension_part can only be provided if the parent type is a tagged type. We give the last sentence anyway for completeness. 5.a.1/2

Implementation Note: We allow a record extension to inherit discriminants; an early version of Ada 9X did not. If the parent subtype is unconstrained, it can be implemented as though its discriminants were repeated in a new known_discriminant_part and then used to constrain the old ones one-for-one. However, in an extension aggregate, the discriminants in this case do not appear in the component association list. 5.a

Ramification: {*AI95-00114-01*} This rule needs to be rechecked in the visible part of an instance of a generic unit because of the “only if” part of the rule. For example: 5.b/2

5.c/2
```
generic
   type T is private;
package P is
   type Der is new T;
end P;
```

5.d/2
```
package I is new P (Some_Tagged_Type); -- illegal
```

{*AI95-00114-01*} The instantiation is illegal because a tagged type is being extended in the visible part without a record_extension_part. Note that this is legal in the private part or body of an instance, both to avoid a contract model violation, and because no code that can see that the type is actually tagged can also see the derived type declaration. 5.e/2

No recheck is needed for derived types with a record_extension_part, as that has to be derived from something that is known to be tagged (otherwise the template is illegal). 5.f/2

{*AI95-00419-01*} If the reserved word **limited** appears in a derived_type_definition, the parent type shall be a limited type. 5.1/2

Reason: We allow **limited** because we don't inherit limitedness from interfaces, so we must have a way to derive a limited type from interfaces. The word **limited** has to be legal when the parent *could be* an interface, and that includes 5.g/2 generic formal abstract types. Since we have to allow it in this case, we might as well allow it everywhere as documentation, to make it explicit that the type is limited.

5.h/2 However, we do not want to allow **limited** when the parent is nonlimited: limitedness cannot change in a derivation tree.

Static Semantics

6 {*constrained (subtype)*} {*unconstrained (subtype)*} The first subtype of the derived type is unconstrained if a known_discriminant_part is provided in the declaration of the derived type, or if the parent subtype is unconstrained. {*corresponding constraint*} Otherwise, the constraint of the first subtype *corresponds* to that of the parent subtype in the following sense: it is the same as that of the parent subtype except that for a range constraint (implicit or explicit), the value of each bound of its range is replaced by the corresponding value of the derived type.

6.a **Discussion:** A digits_constraint in a subtype_indication for a decimal fixed point subtype always imposes a range constraint, implicitly if there is no explicit one given. See 3.5.9, “Fixed Point Types”.

6.1/2 {*AI95-00231-01*} The first subtype of the derived type excludes null (see 3.10) if and only if the parent subtype excludes null.

7 The characteristics of the derived type are defined as follows:

8/2
- {*AI95-00251-01*} {*AI95-00401-01*} {*AI95-00442-01*} [If the parent type or a progenitor type belongs to a class of types, then the derived type also belongs to that class.] The following sets of types, as well as any higher-level sets composed from them, are classes in this sense[, and hence the characteristics defining these classes are inherited by derived types from their parent or progenitor types]: signed integer, modular integer, ordinary fixed, decimal fixed, floating point, enumeration, boolean, character, access-to-constant, general access-to-variable, pool-specific access-to-variable, access-to-subprogram, array, string, non-array composite, nonlimited, untagged record, tagged, task, protected, and synchronized tagged.

8.a **Discussion:** This is inherent in our notion of a “class” of types. It is not mentioned in the initial definition of “class” since at that point type derivation has not been defined. In any case, this rule ensures that every class of types is closed under derivation.

9
- If the parent type is an elementary type or an array type, then the set of possible values of the derived type is a copy of the set of possible values of the parent type. For a scalar type, the base range of the derived type is the same as that of the parent type.

9.a **Discussion:** The base range of a type defined by an integer_type_definition or a real_type_definition is determined by the _definition, and is not necessarily the same as that of the corresponding root numeric type from which the newly defined type is implicitly derived. Treating numerics types as implicitly derived from one of the two root numeric types is simply to link them into a type hierarchy; such an implicit derivation does not follow all the rules given here for an explicit derived_type_definition.

10
- If the parent type is a composite type other than an array type, then the components, protected subprograms, and entries that are declared for the derived type are as follows:

11
  - The discriminants specified by a new known_discriminant_part, if there is one; otherwise, each discriminant of the parent type (implicitly declared in the same order with the same specifications) — {*inherited discriminant*} {*inherited component*} in the latter case, the discriminants are said to be *inherited*, or if unknown in the parent, are also unknown in the derived type;

12
  - Each nondiscriminant component, entry, and protected subprogram of the parent type, implicitly declared in the same order with the same declarations; {*inherited component*} {*inherited protected subprogram*} {*inherited entry*} these components, entries, and protected subprograms are said to be *inherited*;

12.a **Ramification:** The profiles of entries and protected subprograms do not change upon type derivation, although the type of the “implicit” parameter identified by the prefix of the name in a call does.

To be honest: Any name in the parent type_declaration that denotes the current instance of the type is replaced with a name denoting the current instance of the derived type, converted to the parent type. 12.b

- Each component declared in a record_extension_part, if any. 13

Declarations of components, protected subprograms, and entries, whether implicit or explicit, occur immediately within the declarative region of the type, in the order indicated above, following the parent subtype_indication. 14

Discussion: The order of declarations within the region matters for record_aggregates and extension_aggregates. 14.a

Ramification: In most cases, these things are implicitly declared *immediately* following the parent subtype_indication. However, 7.3.1, “Private Operations” defines some cases in which they are implicitly declared later, and some cases in which the are not declared at all. 14.b

Discussion: The place of the implicit declarations of inherited components matters for visibility — they are not visible in the known_discriminant_part nor in the parent subtype_indication, but are usually visible within the record_extension_part, if any (although there are restrictions on their use). Note that a discriminant specified in a new known_discriminant_part is not considered “inherited” even if it has the same name and subtype as a discriminant of the parent type. 14.c

- *This paragraph was deleted.*{*AI95-00419-01*} 15/2
- [For each predefined operator of the parent type, there is a corresponding predefined operator of the derived type.] 16

Proof: This is a ramification of the fact that each class that includes the parent type also includes the derived type, and the fact that the set of predefined operators that is defined for a type, as described in 4.5, is determined by the classes to which it belongs. 16.a

Reason: Predefined operators are handled separately because they follow a slightly different rule than user-defined primitive subprograms. In particular the systematic replacement described below does not apply fully to the relational operators for Boolean and the exponentiation operator for Integer. The relational operators for a type derived from Boolean still return Standard.Boolean. The exponentiation operator for a type derived from Integer still expects Standard.Integer for the right operand. In addition, predefined operators "reemerge" when a type is the actual type corresponding to a generic formal type, so they need to be well defined even if hidden by user-defined primitive subprograms. 16.b

- {*AI95-00401-01*} {*inherited subprogram*} For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent type or of a progenitor type that already exists at the place of the derived_type_definition, there exists a corresponding *inherited* primitive subprogram of the derived type with the same defining name. {*equality operator (special inheritance rule for tagged types)*} Primitive user-defined equality operators of the parent type and any progenitor types are also inherited by the derived type, except when the derived type is a nonlimited record extension, and the inherited operator would have a profile that is type conformant with the profile of the corresponding predefined equality operator; in this case, the user-defined equality operator is not inherited, but is rather incorporated into the implementation of the predefined equality operator of the record extension (see 4.5.2). {*type conformance* [partial]} 17/2

Ramification: We say “...already exists...” rather than “is visible” or “has been declared” because there are certain operations that are declared later, but still exist at the place of the derived_type_definition, and there are operations that are never declared, but still exist. These cases are explained in 7.3.1. 17.a

Note that nonprivate extensions can appear only after the last primitive subprogram of the parent — the freezing rules ensure this. 17.b

Reason: A special case is made for the equality operators on nonlimited record extensions because their predefined equality operators are already defined in terms of the primitive equality operator of their parent type (and of the tagged components of the extension part). Inheriting the parent's equality operator as is would be undesirable, because it would ignore any components of the extension part. On the other hand, if the parent type is limited, then any user-defined equality operator is inherited as is, since there is no predefined equality operator to take its place. 17.c

Ramification: {*AI95-00114-01*} Because user-defined equality operators are not inherited by nonlimited record extensions, the formal parameter names of = and /= revert to Left and Right, even if different formal parameter names were used in the user-defined equality operators of the parent type. 17.d/2

Discussion: {*AI95-00401-01*} This rule only describes what operations are inherited; the rules that describe what happens when there are conflicting inherited subprograms are found in 8.3. 17.e/2

18/2 {*AI95-00401-01*} The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent or progenitor type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent or progenitor type with a *corresponding subtype* of the derived type. {*corresponding subtype*} For a given subtype of the parent or progenitor type, the corresponding subtype of the derived type is defined as follows:

19 • If the declaration of the derived type has neither a known_discriminant_part nor a record_extension_part, then the corresponding subtype has a constraint that corresponds (as defined above for the first subtype of the derived type) to that of the given subtype.

20 • If the derived type is a record extension, then the corresponding subtype is the first subtype of the derived type.

21 • If the derived type has a new known_discriminant_part but is not a record extension, then the corresponding subtype is constrained to those values that when converted to the parent type belong to the given subtype (see 4.6). {*implicit subtype conversion (derived type discriminants)* [partial]}

21.a **Reason:** An inherited subprogram of an untagged type has an Intrinsic calling convention, which precludes the use of the Access attribute. We preclude 'Access because correctly performing all required constraint checks on an indirect call to such an inherited subprogram was felt to impose an undesirable implementation burden.

22/2 {*AI95-00401-01*} The same formal parameters have default_expressions in the profile of the inherited subprogram. [Any type mismatch due to the systematic replacement of the parent or progenitor type by the derived type is handled as part of the normal type conversion associated with parameter passing — see 6.4.1.]

22.a/2 **Reason:** {*AI95-00401-01*} We don't introduce the type conversion explicitly here since conversions to record extensions or on access parameters are not generally legal. Furthermore, any type conversion would just be "undone" since the subprogram of the parent or progenitor is ultimately being called anyway. (Null procedures can be inherited from a progenitor without being overridden, so it is possible to call subprograms of an interface.)

23/2 {*AI95-00401-01*} If a primitive subprogram of the parent or progenitor type is visible at the place of the derived_type_definition, then the corresponding inherited subprogram is implicitly declared immediately after the derived_type_definition. Otherwise, the inherited subprogram is implicitly declared later or not at all, as explained in 7.3.1.

24 {*derived type* [partial]} A derived type can also be defined by a private_extension_declaration (see 7.3) or a formal_derived_type_definition (see 12.5.1). Such a derived type is a partial view of the corresponding full or actual type.

25 All numeric types are derived types, in that they are implicitly derived from a corresponding root numeric type (see 3.5.4 and 3.5.6).

Dynamic Semantics

26 {*elaboration (derived_type_definition)* [partial]} The elaboration of a derived_type_definition creates the derived type and its first subtype, and consists of the elaboration of the subtype_indication and the record_extension_part, if any. If the subtype_indication depends on a discriminant, then only those expressions that do not depend on a discriminant are evaluated.

26.a/2 **Discussion:** {*AI95-00251-01*} We don't mention the interface_list, because it does not need elaboration (see 3.9.4. This is consistent with the handling of discriminant_parts, which aren't elaborated either.

27/2 {*AI95-00391-01*} {*AI95-00401-01*} {*execution (call on an inherited subprogram)* [partial]} For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent or progenitor type is performed; the normal conversion of each actual parameter to the subtype of the corresponding formal parameter (see 6.4.1) performs any necessary type conversion as well. If the result type of the inherited subprogram is the derived type, the result of calling the subprogram of the parent or progenitor is converted to the derived type, or in the case of a null extension, extended to the derived type using the equivalent of an extension_aggregate with the original result as the ancestor_part and **null record** as the record_component_association_list. {*implicit subtype conversion (result of inherited function)* [partial]}

Discussion: {*AI95-00391-01*} If an inherited function returns the derived type, and the type is a non-null record extension, then the inherited function shall be overridden, unless the type is abstract (in which case the function is abstract, and (unless overridden) cannot be called except via a dispatching call). See 3.9.3. 27.a/2

NOTES

10 {*closed under derivation*} Classes are closed under derivation — any class that contains a type also contains its derivatives. Operations available for a given class of types are available for the derived types in that class. 28

11 Evaluating an inherited enumeration literal is equivalent to evaluating the corresponding enumeration literal of the parent type, and then converting the result to the derived type. This follows from their equivalence to parameterless functions. {*implicit subtype conversion (inherited enumeration literal)* [partial]} 29

12 A generic subprogram is not a subprogram, and hence cannot be a primitive subprogram and cannot be inherited by a derived type. On the other hand, an instance of a generic subprogram can be a primitive subprogram, and hence can be inherited. 30

13 If the parent type is an access type, then the parent and the derived type share the same storage pool; there is a **null** access value for the derived type and it is the implicit initial value for the type. See 3.10. 31

14 If the parent type is a boolean type, the predefined relational operators of the derived type deliver a result of the predefined type Boolean (see 4.5.2). If the parent type is an integer type, the right operand of the predefined exponentiation operator is of the predefined type Integer (see 4.5.6). 32

15 Any discriminants of the parent type are either all inherited, or completely replaced with a new set of discriminants. 33

16 For an inherited subprogram, the subtype of a formal parameter of the derived type need not have any value in common with the first subtype of the derived type. 34

Proof: This happens when the parent subtype is constrained to a range that does not overlap with the range of a subtype of the parent type that appears in the profile of some primitive subprogram of the parent type. For example: 34.a

34.b
```
type T1 is range 1..100;
subtype S1 is T1 range 1..10;
procedure P(X : in S1);  -- P is a primitive subprogram
type T2 is new T1 range 11..20;
-- implicitly declared:
-- procedure P(X : in T2'Base range 1..10);
--      X cannot be in T2'First .. T2'Last
```

17 If the reserved word **abstract** is given in the declaration of a type, the type is abstract (see 3.9.3). 35

18 {*AI95-00251-01*} {*AI95-00401-01*} An interface type that has a progenitor type “is derived from” that type. A derived_type_definition, however, never defines an interface type. 35.1/2

19 {*AI95-00345-01*} It is illegal for the parent type of a derived_type_definition to be a synchronized tagged type. 35.2/2

Proof: 3.9.1 prohibits record extensions whose parent type is a synchronized tagged type, and this clause requires tagged types to have a record extension. Thus there are no legal derivations. Note that a synchronized interface can be used as a progenitor in an interface_type_definition as well as in task and protected types, but we do not allow concrete extensions of any synchronized tagged type. 35.a/2

Examples

Examples of derived type declarations: 36

37
```
type Local_Coordinate is new Coordinate;   -- two different types
type Midweek is new Day range Tue .. Thu;  -- see 3.5.1
type Counter is new Positive;              -- same range as Positive
```

38
```
type Special_Key is new Key_Manager.Key;   -- see 7.3.1
  -- the inherited subprograms have the following specifications:
  --         procedure Get_Key(K : out Special_Key);
  --         function "<"(X,Y : Special_Key) return Boolean;
```

Inconsistencies With Ada 83

{*inconsistencies with Ada 83*} When deriving from a (nonprivate, nonderived) type in the same visible part in which it is defined, if a predefined operator had been overridden prior to the derivation, the derived type will inherit the user- 38.a defined operator rather than the predefined operator. The work-around (if the new behavior is not the desired behavior) is to move the definition of the derived type prior to the overriding of any predefined operators.

Incompatibilities With Ada 83

38.b {*incompatibilities with Ada 83*} When deriving from a (nonprivate, nonderived) type in the same visible part in which it is defined, a primitive subprogram of the parent type declared before the derived type will be inherited by the derived type. This can cause upward incompatibilities in cases like this:

38.c
```
      package P is
         type T is (A, B, C, D);
         function F( X : T := A ) return Integer;
         type NT is new T;
         -- inherits F as
         -- function F( X : NT := A ) return Integer;
         -- in Ada 95 only
         ...
      end P;
      ...
      use P;   -- Only one declaration of F from P is use-visible in
               -- Ada 83;  two declarations of F are use-visible in
               -- Ada 95.
   begin
      ...
      if F > 1 then ... -- legal in Ada 83, ambiguous in Ada 95
```

Extensions to Ada 83

38.d {*extensions to Ada 83*} The syntax for a derived_type_definition is amended to include an optional record_extension_part (see 3.9.1).

38.e A derived type may override the discriminants of the parent by giving a new discriminant_part.

38.f The parent type in a derived_type_definition may be a derived type defined in the same visible part.

38.g When deriving from a type in the same visible part in which it is defined, the primitive subprograms declared prior to the derivation are inherited as primitive subprograms of the derived type. See 3.2.3.

Wording Changes from Ada 83

38.h We now talk about the classes to which a type belongs, rather than a single class.

38.i As explained in Section 13, the concept of "storage pool" replaces the Ada 83 concept of "collection." These concepts are similar, but not the same.

Extensions to Ada 95

38.j/2 {*AI95-00251-01*} {*AI95-00401-01*} {*extensions to Ada 95*} A derived type may inherit from multiple (interface) progenitors, as well as the parent type — see 3.9.4, "Interface Types".

38.k/2 {*AI95-00419-01*} A derived type may specify that it is a limited type. This is required for interface ancestors (from which limitedness is not inherited), but it is generally useful as documentation of limitedness.

Wording Changes from Ada 95

38.l/2 {*AI95-00391-01*} Defined the result of functions for null extensions (which we no longer require to be overridden - see 3.9.3).

38.m/2 {*AI95-00442-01*} Defined the term "category of types" and used it in wording elsewhere; also specified the language-defined categories that form classes of types (this was never normatively specified in Ada 95.

3.4.1 Derivation Classes

1 In addition to the various language-defined classes of types, types can be grouped into *derivation classes*.

Static Semantics

2/2 {*AI95-00251-01*} {*AI95-00401-01*} {*derived from (directly or indirectly)*} A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. A derived type, interface type, type extension, task type, protected type, or formal derived type is also derived from every ancestor of each of its progenitor types, if any. {*derivation class (for a type)*} {*root type (of a class)*} {*rooted at a type*} The derivation class of types for a type *T* (also called the class *rooted* at *T*) is the set consisting of *T* (the *root type* of the class) and all types derived from *T* (directly or indirectly) plus any associated universal or class-wide types (defined below).

2.a **Discussion:** Note that the definition of "derived from" is a recursive definition. We don't define a root type for all interesting language-defined classes, though presumably we could.

2.b **To be honest:** By the class-wide type "associated" with a type *T*, we mean the type *T*'Class. Similarly, the universal type associated with *root_integer*, *root_real*, and *root_fixed* are *universal_integer*, *universal_real*, and *universal_fixed*, respectively.

3/2 {*AI95-00230-01*} Every type is either a *specific* type, a *class-wide* type, or a *universal* type. {*specific type*} A specific type is one defined by a type_declaration, a formal_type_declaration, or a full type definition embedded in another construct. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

3.a **To be honest:** The root types *root_integer*, *root_real*, and *root_fixed* are also specific types. They are declared in the specification of package Standard.

4 {*class-wide type*} Class-wide types

Class-wide types are defined for [(and belong to)] each derivation class rooted at a tagged type (see 3.9). Given a subtype S of a tagged type *T*, S'Class is the subtype_mark for a corresponding subtype of the tagged class-wide type *T*'Class. Such types are called "class-wide" because when a formal parameter is defined to be of a class-wide type *T*'Class, an actual parameter of any type in the derivation class rooted at *T* is acceptable (see 8.6).

5 {*first subtype*} The set of values for a class-wide type *T*'Class is the discriminated union of the set of values of each specific type in the derivation class rooted at *T* (the tag acts as the implicit discriminant — see 3.9). Class-wide types have no primitive subprograms of their own. However, as explained in 3.9.2, operands of a class-wide type *T*'Class can be used as part of a dispatching call on a primitive subprogram of the type *T*. The only components [(including discriminants)] of *T*'Class that are visible are those of *T*. If S is a first subtype, then S'Class is a first subtype.

5.a **Reason:** We want S'Class to be a first subtype when S is, so that an attribute_definition_clause like "**for** S'Class'Output **use** ...;" will be legal.

6/2 {*AI95-00230-01*} {*universal type*} Universal types

Universal types are defined for [(and belong to)] the integer, real, fixed point, and access classes, and are referred to in this standard as respectively, *universal_integer*, *universal_real*, *universal_fixed*, and *universal_access*. These are analogous to class-wide types for these language-defined elementary classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the corresponding class is acceptable. In addition, a value of a universal type (including an integer or real numeric_literal, or the literal **null**) is "universal" in that it is acceptable where some particular type in the class is expected (see 8.6).

7 The set of values of a universal type is the undiscriminated union of the set of values possible for any definable type in the associated class. Like class-wide types, universal types have no primitive subprograms of their own. However, their "universality" allows them to be used as operands with the primitive subprograms of any type in the corresponding class.

7.a **Discussion:** A class-wide type is only class-wide in one direction, from specific to class-wide, whereas a universal type is class-wide (universal) in both directions, from specific to universal and back.

7.b/2 {*AI95-00230-01*} We considered defining class-wide or perhaps universal types for all derivation classes, not just tagged classes and these four elementary classes. However, this was felt to overly weaken the strong-typing model in some situations. Tagged types preserve strong type distinctions thanks to the run-time tag. Class-wide or universal types for untagged types would weaken the compile-time type distinctions without providing a compensating run-time-checkable distinction.

7.c We considered defining standard names for the universal numeric types so they could be used in formal parameter specifications. However, this was felt to impose an undue implementation burden for some implementations.

7.d **To be honest:** Formally, the set of values of a universal type is actually a *copy* of the undiscriminated union of the values of the types in its class. This is because we want each value to have exactly one type, with explicit or implicit conversion needed to go between types. An alternative, consistent model would be to associate a class, rather than a particular type, with a value, even though any given expression would have a particular type. In that case, implicit type conversions would not generally need to change the value, although an associated subtype conversion might need to.

8 {*root_integer* [partial]} {*root_real* [partial]} The integer and real numeric classes each have a specific root type in addition to their universal type, named respectively *root_integer* and *root_real*.

9 {*cover (a type)*} A class-wide or universal type is said to *cover* all of the types in its class. A specific type covers only itself.

10/2 {*AI95-00230-01*} {*AI95-00251-01*} {*descendant (of a type)*} A specific type *T2* is defined to be a *descendant* of a type *T1* if *T2* is the same as *T1*, or if *T2* is derived (directly or indirectly) from *T1*. A class-wide type *T2*'Class is defined to be a descendant of type *T1* if *T2* is a descendant of *T1*. Similarly, the numeric universal types are defined to be descendants of the root types of their classes. {*ancestor (of a type)*} If a type *T2* is a descendant of a type *T1*, then *T1* is called an *ancestor* of *T2*. {*ultimate ancestor (of a type)*} {*ancestor (ultimate)*} An *ultimate ancestor* of a type is an ancestor of that type that is not itself a descendant of any other type. Every untagged type has a unique ultimate ancestor.

10.a **Ramification:** A specific type is a descendant of itself. Class-wide types are considered descendants of the corresponding specific type, and do not have any descendants of their own.

10.b A specific type is an ancestor of itself. The root of a derivation class is an ancestor of all types in the class, including any class-wide types in the class.

10.c **Discussion:** The terms root, parent, ancestor, and ultimate ancestor are all related. For example:

10.d/2
- {*AI95-00251-01*} Each type has at most one parent, and one or more ancestor types; each untagged type has exactly one ultimate ancestor. In Ada 83, the term “parent type” was sometimes used more generally to include any ancestor type (e.g. RM83-9.4(14)). In Ada 95, we restrict parent to mean the immediate ancestor.

10.e
- A class of types has at most one root type; a derivation class has exactly one root type.

10.f
- The root of a class is an ancestor of all of the types in the class (including itself).

10.g
- The type *root_integer* is the root of the integer class, and is the ultimate ancestor of all integer types. A similar statement applies to *root_real*.

10.h/2 **Glossary entry:** {*Ancestor*} An ancestor of a type is the type itself or, in the case of a type derived from other types, its parent type or one of its progenitor types or one of their ancestors. Note that ancestor and descendant are inverse relationships.

10.i/2 **Glossary entry:** {*Descendant*} A type is a descendant of itself, its parent and progenitor types, and their ancestors. Note that descendant and ancestor are inverse relationships.

11 {*inherited (from an ancestor type)*} An inherited component [(including an inherited discriminant)] of a derived type is inherited *from* a given ancestor of the type if the corresponding component was inherited by each derived type in the chain of derivations going back to the given ancestor.

NOTES

12 20 Because operands of a universal type are acceptable to the predefined operators of any type in their class, ambiguity can result. For *universal_integer* and *universal_real*, this potential ambiguity is resolved by giving a preference (see 8.6) to the predefined operators of the corresponding root types (*root_integer* and *root_real*, respectively). Hence, in an apparently ambiguous expression like

13
```
1 + 4 < 7
```

14 where each of the literals is of type *universal_integer*, the predefined operators of *root_integer* will be preferred over those of other specific integer types, thereby resolving the ambiguity.

14.a **Ramification:** Except for this preference, a root numeric type is essentially like any other specific type in the associated numeric class. In particular, the result of a predefined operator of a root numeric type is not “universal” (implicitly convertible) even if both operands were.

Wording Changes from Ada 95

14.b/2 {*AI95-00230-01*} Updated the wording to define the *universal_access* type. This was defined to make **null** for anonymous access types sensible.

{*AI95-00251-01*} {*AI95-00401-01*} The definitions of ancestors and descendants were updated to allow multiple ancestors (necessary to support interfaces). 14.c/2

3.5 Scalar Types

1 {*scalar type*} *Scalar* types comprise enumeration types, integer types, and real types. {*discrete type*} Enumeration types and integer types are called *discrete* types; {*position number*} each value of a discrete type has a *position number* which is an integer value. {*numeric type*} Integer types and real types are called *numeric* types. [All scalar types are ordered, that is, all relational operators are predefined for their values.]

Syntax

2 range_constraint ::= **range** range

3 range ::= range_attribute_reference
  | simple_expression .. simple_expression

3.a **Discussion:** These need to be simple_expressions rather than more general expressions because ranges appear in membership tests and other contexts where expression .. expression would be ambiguous.

4 {*range*} {*lower bound (of a range)*} {*upper bound (of a range)*} {*type of a range*} A *range* has a *lower bound* and an *upper bound* and specifies a subset of the values of some scalar type (the *type of the range*). A range with lower bound L and upper bound R is described by “L .. R”. {*null range*} If R is less than L, then the range is a *null range*, and specifies an empty set of values. Otherwise, the range specifies the values of the type from the lower bound to the upper bound, inclusive. {*belong (to a range)*} A value *belongs* to a range if it is of the type of the range, and is in the subset of values specified by the range. {*satisfies (a range constraint)* [partial]} A value *satisfies* a range constraint if it belongs to the associated range. {*included (one range in another)*} One range is *included* in another if all values that belong to the first range also belong to the second.

Name Resolution Rules

5 {*expected type (range_constraint range)* [partial]} For a subtype_indication containing a range_constraint, either directly or as part of some other scalar_constraint, the type of the range shall resolve to that of the type determined by the subtype_mark of the subtype_indication. {*expected type (range simple_expressions)* [partial]} For a range of a given type, the simple_expressions of the range (likewise, the simple_expressions of the equivalent range for a range_attribute_reference) are expected to be of the type of the range.

5.a **Discussion:** In Ada 95, constraints only appear within subtype_indications; things that look like constraints that appear in type declarations are called something else like real_range_specifications.

5.b We say "the expected type is ..." or "the type is expected to be ..." depending on which reads better. They are fundamentally equivalent, and both feed into the type resolution rules of clause 8.6.

5.c In some cases, it doesn't work to use expected types. For example, in the above rule, we say that the “type of the range shall resolve to ...” rather than “the expected type for the range is ...”. We then use “expected type” for the bounds. If we used “expected” at both points, there would be an ambiguity, since one could apply the rules of 8.6 either on determining the type of the range, or on determining the types of the individual bounds. It is clearly important to allow one bound to be of a universal type, and the other of a specific type, so we need to use “expected type” for the bounds. Hence, we used “shall resolve to” for the type of the range as a whole. There are other situations where “expected type” is not quite right, and we use “shall resolve to” instead.

Static Semantics

6 {*base range (of a scalar type)* [distributed]} The *base range* of a scalar type is the range of finite values of the type that can be represented in every unconstrained object of the type; it is also the range supported at a minimum for intermediate values during the evaluation of expressions involving predefined operators of the type.

6.a **Implementation Note:** Note that in some machine architectures intermediates in an expression (particularly if static), and register-resident variables might accommodate a wider range. The base range does not include the values of this wider range that are not assignable without overflow to memory-resident objects.

6.b **Ramification:** {*base range (of an enumeration type)* [partial]} The base range of an enumeration type is the range of values of the enumeration type.

6.c **Reason:** If the representation supports infinities, the base range is nevertheless restricted to include only the representable finite values, so that 'Base'First and 'Base'Last are always guaranteed to be finite.

6.d **To be honest:** By a "value that can be assigned without overflow" we don't mean to restrict ourselves to values that can be represented exactly. Values between machine representable values can be assigned, but on subsequent reading, a slightly different value might be retrieved, as (partially) determined by the number of digits of precision of the type.

7 {*constrained (subtype)*} {*unconstrained (subtype)*} [A constrained scalar subtype is one to which a range constraint applies.] {*range (of a scalar subtype)*} The *range* of a constrained scalar subtype is the range associated with the range constraint of the subtype. The *range* of an unconstrained scalar subtype is the base range of its type.

Dynamic Semantics

8 {*compatibility (range with a scalar subtype)* [partial]} A range is *compatible* with a scalar subtype if and only if it is either a null range or each bound of the range belongs to the range of the subtype. {*compatibility (range_constraint with a scalar subtype)* [partial]} A range_constraint is *compatible* with a scalar subtype if and only if its range is compatible with the subtype.

8.a **Ramification:** Only range_constraints (explicit or implicit) impose conditions on the values of a scalar subtype. The other scalar_constraints, digits_constraints and delta_constraints impose conditions on the subtype denoted by the subtype_mark in a subtype_indication, but don't impose a condition on the values of the subtype being defined. Therefore, a scalar subtype is not called *constrained* if all that applies to it is a digits_constraint. Decimal subtypes are subtle, because a digits_constraint without a range_constraint nevertheless includes an implicit range_constraint.

9 {*elaboration (range_constraint)* [partial]} The elaboration of a range_constraint consists of the evaluation of the range. {*evaluation (range)* [partial]} The evaluation of a range determines a lower bound and an upper bound. If simple_expressions are given to specify bounds, the evaluation of the range evaluates these simple_expressions in an arbitrary order, and converts them to the type of the range. {*implicit subtype conversion (bounds of a range)* [partial]} If a range_attribute_reference is given, the evaluation of the range consists of the evaluation of the range_attribute_reference.

10 *Attributes*

11 For every scalar subtype S, the following attributes are defined:

12 S'First S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S.

12.a **Ramification:** Evaluating S'First never raises Constraint_Error.

13 S'Last S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S.

13.a **Ramification:** Evaluating S'Last never raises Constraint_Error.

14 S'Range S'Range is equivalent to the range S'First .. S'Last.

15 S'Base S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the *base subtype* of the type. {*base subtype (of a type)*}

16 S'Min S'Min denotes a function with the following specification:

17
```
function S'Min(Left, Right : S'Base)
  return S'Base
```

18 The function returns the lesser of the values of the two parameters.

18.a **Discussion:** {*italics (formal parameters of attribute functions)*} The formal parameter names are italicized because they cannot be used in calls — see 6.4. Such a specification cannot be written by the user because an attribute_reference is not permitted as the designator of a user-defined function, nor can its formal parameters be anonymous.

S'Max S'Max denotes a function with the following specification: 19

```
function S'Max(Left, Right : S'Base)
  return S'Base
```
20

The function returns the greater of the values of the two parameters. 21

S'Succ S'Succ denotes a function with the following specification: 22

```
function S'Succ(Arg : S'Base)
  return S'Base
```
23

{*Constraint_Error (raised by failure of run-time check)*} For an enumeration type, the function returns the value whose position number is one more than that of the value of *Arg*; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of adding one to the value of *Arg*. For a fixed point type, the function returns the result of adding *small* to the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately above the value of *Arg*; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if there is no such machine number. 24

Ramification: S'Succ for a modular integer subtype wraps around if the value of *Arg* is S'Base'Last. S'Succ for a signed integer subtype might raise Constraint_Error if the value of *Arg* is S'Base'Last, or it might return the out-of-base-range value S'Base'Last+1, as is permitted for all predefined numeric operations. 24.a

S'Pred S'Pred denotes a function with the following specification: 25

```
function S'Pred(Arg : S'Base)
  return S'Base
```
26

{*Constraint_Error (raised by failure of run-time check)*} For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of *Arg*. For a fixed point type, the function returns the result of subtracting *small* from the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of *Arg*; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if there is no such machine number. 27

Ramification: S'Pred for a modular integer subtype wraps around if the value of *Arg* is S'Base'First. S'Pred for a signed integer subtype might raise Constraint_Error if the value of *Arg* is S'Base'First, or it might return the out-of-base-range value S'Base'First–1, as is permitted for all predefined numeric operations. 27.a

S'Wide_Wide_Image 27.1/2
{*AI95-00285-01*} S'Wide_Wide_Image denotes a function with the following specification:

```
function S'Wide_Wide_Image(Arg : S'Base)
  return Wide_Wide_String
```
27.2/2

{*image (of a value)*} The function returns an *image* of the value of *Arg*, that is, a sequence of characters representing the value in display form. The lower bound of the result is one. 27.3/2

The image of an integer value is the corresponding decimal literal, without underlines, leading zeros, exponent, or trailing spaces, but with a single leading character that is either a minus sign or a space. 27.4/2

Implementation Note: If the machine supports negative zeros for signed integer types, it is not specified whether " 0" or "–0" should be returned for negative zero. We don't have enough experience with such machines to know what is appropriate, and what other languages do. In any case, the implementation should be consistent. 27.b/2

{*nongraphic character*} The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); 27.5/2 neither leading nor trailing spaces are included. For a *nongraphic character* (a value of a character type that has no enumeration literal associated with it), the result is a corresponding language-defined name in upper case (for example, the image of the nongraphic character identified as *nul* is “NUL” — the quotes are not part of the image).

27.c/2 **Implementation Note:** For an enumeration type T that has “holes” (caused by an enumeration_representation_clause), {*Program_Error (raised by failure of run-time check)*} T'Wide_Image should raise Program_Error if the value is one of the holes (which is a bounded error anyway, since holes can be generated only via uninitialized variables and similar things.

27.6/2 The image of a floating point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, a single digit (that is nonzero unless the value is zero), a decimal point, S'Digits–1 (see 3.5.8) digits after the decimal point (but one if S'Digits is one), an upper case E, the sign of the exponent (either + or –), and two or more digits (with leading zeros if necessary) representing the exponent. If S'Signed_Zeros is True, then the leading character is a minus sign for a negatively signed zero.

27.d/2 **To be honest:** Leading zeros are present in the exponent only if necessary to make the exponent at least two digits.

27.e/2 **Reason:** This image is intended to conform to that produced by Text_IO.Float_IO.Put in its default format.

27.f/2 **Implementation Note:** The rounding direction is specified here to ensure portability of output results.

27.7/2 The image of a fixed point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, one or more digits before the decimal point (with no redundant leading zeros), a decimal point, and S'Aft (see 3.5.10) digits after the decimal point.

27.g/2 **Reason:** This image is intended to conform to that produced by Text_IO.Fixed_IO.Put.

27.h/2 **Implementation Note:** The rounding direction is specified here to ensure portability of output results.

27.i/2 **Implementation Note:** For a machine that supports negative zeros, it is not specified whether " 0.000" or "–0.000" is returned. See corresponding comment above about integer types with signed zeros.

28 S'Wide_Image
S'Wide_Image denotes a function with the following specification:

29
```
function S'Wide_Image(Arg : S'Base)
  return Wide_String
```

30/2 {*AI95-00285-01*} {*image (of a value)*} The function returns an image of the value of *Arg* as a Wide_String. The lower bound of the result is one. The image has the same sequence of character as defined for S'Wide_Wide_Image if all the graphic characters are defined in Wide_Character; otherwise the sequence of characters is implementation defined (but no shorter than that of S'Wide_Wide_Image for the same value of Arg).

30.a/2 **Implementation defined:** The sequence of characters of the value returned by S'Wide_Image when some of the graphic characters of S'Wide_Wide_Image are not defined in Wide_Character.

Paragraphs 31 through 34 were moved to Wide_Wide_Image.

35 S'Image S'Image denotes a function with the following specification:

36
```
function S'Image(Arg : S'Base)
  return String
```

37/2 {*AI95-00285-01*} The function returns an image of the value of *Arg* as a String. The lower bound of the result is one. The image has the same sequence of graphic characters as that defined for S'Wide_Wide_Image if all the graphic characters are defined in Character; otherwise the sequence of characters is implementation defined (but no shorter than that of S'Wide_Wide_Image for the same value of *Arg*).

37.a/2 **Implementation defined:** The sequence of characters of the value returned by S'Image when some of the graphic characters of S'Wide_Wide_Image are not defined in Character.

S'Wide_Wide_Width 37.1/2
{*AI95-00285-01*} S'Wide_Wide_Width denotes the maximum length of a Wide_Wide_String returned by S'Wide_Wide_Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

S'Wide_Width 38
S'Wide_Width denotes the maximum length of a Wide_String returned by S'Wide_Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

S'Width S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. 39

S'Wide_Wide_Value 39.1/2
{*AI95-00285-01*} S'Wide_Wide_Value denotes a function with the following specification:

39.2/2
```
function S'Wide_Wide_Value(Arg : Wide_Wide_String)
  return S'Base
```

39.3/2 This function returns a value given an image of the value as a Wide_Wide_String, ignoring any leading or trailing spaces.

39.4/2 {*evaluation (Wide_Wide_Value)* [partial]} {*Constraint_Error (raised by failure of run-time check)*} For the evaluation of a call on S'Wide_Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Wide_Image for a nongraphic character of the type), the result is the corresponding enumeration value; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} otherwise Constraint_Error is raised.

39.a.1/2 **Discussion:** It's not crystal clear that Range_Check is appropriate here, but it doesn't seem worthwhile to invent a whole new check name just for this weird case, so we decided to lump it in with Range_Check.

39.a.2/2 **To be honest:** {*8652/0096*} {*AI95-00053-01*} A sequence of characters corresponds to the result of S'Wide_Wide_Image if it is the same ignoring case. Thus, the case of an image of a nongraphic character does not matter. For example, Character'Wide_Wide_Value("nul") does not raise Constraint_Error, even though Character'Wide_Wide_Image returns "NUL" for the nul character.

39.5/2 {*Constraint_Error (raised by failure of run-time check)*} For the evaluation of a call on S'Wide_Wide_Value for an integer subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an integer literal, with an optional leading sign character (plus or minus for a signed type; only plus for a modular type), and the corresponding numeric value belongs to the base range of the type of S, then that value is the result; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} otherwise Constraint_Error is raised.

39.a.3/2 **Discussion:** We considered allowing 'Value to return a representable but out-of-range value without a Constraint_Error. However, we currently require (see 4.9) in an assignment_statement like "X := <numeric_literal>;" that the value of the numeric-literal be in X's base range (at compile time), so it seems unfriendly and confusing to have a different range allowed for 'Value. Furthermore, for modular types, without the requirement for being in the base range, 'Value would have to handle arbitrarily long literals (since overflow never occurs for modular types).

39.6/2 For the evaluation of a call on S'Wide_Wide_Value for a real subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of one of the following:

- numeric_literal 39.7/2
- numeral.[exponent] 39.8/2
- .numeral[exponent] 39.9/2
- base#based_numeral.#[exponent] 39.10/2
- base#.based_numeral#[exponent] 39.11/2

39.12/2 {*Constraint_Error (raised by failure of run-time check)*} with an optional leading sign character (plus or minus), and if the corresponding numeric value belongs to the base range of the type of S, then that value is the result; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} otherwise Constraint_Error is raised. The sign of a zero value is preserved (positive if none has been specified) if S'Signed_Zeros is True.

40 S'Wide_Value

S'Wide_Value denotes a function with the following specification:

41
```
function S'Wide_Value(Arg : Wide_String)
  return S'Base
```

42 This function returns a value given an image of the value as a Wide_String, ignoring any leading or trailing spaces.

43/2 {*AI95-00285-01*} {*evaluation (Wide_Value)* [partial]} {*Constraint_Error (raised by failure of run-time check)*} For the evaluation of a call on S'Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Image for a value of the type), the result is the corresponding enumeration value; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} otherwise Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Wide_Value with *Arg* of type Wide_String is equivalent to a call on S'Wide_Wide_Value for a corresponding *Arg* of type Wide_Wide_String.

43.a/2 *This paragraph was deleted.*

43.a.1/2 *This paragraph was deleted.*{*8652/0096*} {*AI95-00053-01*}

43.b/2 **Reason:** S'Wide_Value is subtly different from S'Wide_Wide_Value for enumeration subtypes since S'Wide_Image might produce a different sequence of characters than S'Wide_Wide_Image if the enumeration literal uses characters outside of the predefined type Wide_Character. That is why we don't just define S'Wide_Value in terms of S'Wide_Wide_Value for enumeration subtypes. S'Wide_Value and S'Wide_Wide_Value for numeric subtypes yield the same result given the same sequence of characters.

Paragraphs 44 through 51 were moved to Wide_Wide_Value.

52 S'Value S'Value denotes a function with the following specification:

53
```
function S'Value(Arg : String)
  return S'Base
```

54 This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces.

55/2 {*AI95-00285-01*} {*evaluation (Value)* [partial]} {*Constraint_Error (raised by failure of run-time check)*} For the evaluation of a call on S'Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Image for a value of the type), the result is the corresponding enumeration value; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} otherwise Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Value with *Arg* of type String is equivalent to a call on S'Wide_Wide_Value for a corresponding *Arg* of type Wide_Wide_String.

55.a/2 **Reason:** {*AI95-00285-01*} S'Value is subtly different from S'Wide_Wide_Value for enumeration subtypes; see the discussion under S'Wide_Value.

Implementation Permissions

56/2 {*AI95-00285-01*} An implementation may extend the Wide_Wide_Value, [Wide_Value, Value, Wide_Wide_Image, Wide_Image, and Image] attributes of a floating point type to support special values such as infinities and NaNs.

Proof: {*AI95-00285-01*} The permission is really only necessary for Wide_Wide_Value, because Value and Wide_Value are defined in terms of Wide_Wide_Value, and because the behavior of Wide_Wide_Image, Wide_Image, and Image is already unspecified for things like infinities and NaNs. 56.a/2

Reason: This is to allow implementations to define full support for IEEE arithmetic. See also the similar permission for Get in A.10.9. 56.b

NOTES

21 The evaluation of S'First or S'Last never raises an exception. If a scalar subtype S has a nonnull range, S'First and S'Last belong to this range. These values can, for example, always be assigned to a variable of subtype S. 57

Discussion: This paragraph addresses an issue that came up with Ada 83, where for fixed point types, the end points of the range specified in the type definition were not necessarily within the base range of the type. However, it was later clarified (and we reconfirm it in 3.5.9, “Fixed Point Types”) that the First and Last attributes reflect the true bounds chosen for the type, not the bounds specified in the type definition (which might be outside the ultimately chosen base range). 57.a

22 For a subtype of a scalar type, the result delivered by the attributes Succ, Pred, and Value might not belong to the subtype; similarly, the actual parameters of the attributes Succ, Pred, and Image need not belong to the subtype. 58

23 For any value V (including any nongraphic character) of an enumeration subtype S, S'Value(S'Image(V)) equals V, as do S'Wide_Value(S'Wide_Image(V)) and S'Wide_Wide_Value(S'Wide_Wide_Image(V)). None of these expressions ever raise Constraint_Error. 59

Examples

Examples of ranges: 60

61
```
-10 .. 10
X .. X + 1
0.0 .. 2.0*Pi
Red .. Green      -- see 3.5.1
1 .. 0            -- a null range
Table'Range       -- a range attribute reference (see 3.6)
```

Examples of range constraints: 62

63
```
range -999.0 .. +999.0
range S'First+1 .. S'Last-1
```

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} S'Base is no longer defined for nonscalar types. One conceivable existing use of S'Base for nonscalar types is S'Base'Size where S is a generic formal private type. However, that is not generally useful because the actual subtype corresponding to S might be a constrained array or discriminated type, which would mean that S'Base'Size might very well overflow (for example, S'Base'Size where S is a constrained subtype of String will generally be 8 * (Integer'Last + 1)). For derived discriminated types that are packed, S'Base'Size might not even be well defined if the first subtype is constrained, thereby allowing some amount of normally required “dope” to have been squeezed out in the packing. Hence our conclusion is that S'Base'Size is not generally useful in a generic, and does not justify keeping the attribute Base for nonscalar types just so it can be used as a prefix. 63.a/1

Extensions to Ada 83

{*extensions to Ada 83*} The attribute S'Base for a scalar subtype is now permitted anywhere a subtype_mark is permitted. S'Base'First .. S'Base'Last is the base range of the type. Using an attribute_definition_clause, one cannot specify any subtype-specific attributes for the subtype denoted by S'Base (the base subtype). 63.b

The attribute S'Range is now allowed for scalar subtypes. 63.c

The attributes S'Min and S'Max are now defined, and made available for all scalar types. 63.d

The attributes S'Succ, S'Pred, S'Image, S'Value, and S'Width are now defined for real types as well as discrete types. 63.e

Wide_String versions of S'Image and S'Value are defined. These are called S'Wide_Image and S'Wide_Value to avoid introducing ambiguities involving uses of these attributes with string literals. 63.f

Wording Changes from Ada 83

We now use the syntactic category range_attribute_reference since it is now syntactically distinguished from other attribute references. 63.g

The definition of S'Base has been moved here from 3.3.3 since it now applies only to scalar types. 63.h

63.i More explicit rules are provided for nongraphic characters.

Extensions to Ada 95

63.j/2 {*AI95-00285-01*} {*extensions to Ada 95*} The attributes Wide_Wide_Image, Wide_Wide_Value, and Wide_Wide_Width are new. Note that Wide_Image and Wide_Value are now defined in terms of Wide_Wide_Image and Wide_Wide_Value, but the image of types other than characters have not changed.

Wording Changes from Ada 95

63.k/2 {*AI95-00285-01*} The Wide_Image and Wide_Value attributes are now defined in terms of Wide_Wide_Image and Wide_Wide_Value, but the images of numeric types have not changed.

3.5.1 Enumeration Types

1 [{*enumeration type*} An enumeration_type_definition defines an enumeration type.]

Syntax

2 enumeration_type_definition ::=
  (enumeration_literal_specification {, enumeration_literal_specification})

3 enumeration_literal_specification ::= defining_identifier | defining_character_literal

4 defining_character_literal ::= character_literal

Legality Rules

5 [The defining_identifiers and defining_character_literals listed in an enumeration_type_definition shall be distinct.]

5.a **Proof:** This is a ramification of the normal disallowance of homographs explicitly declared immediately in the same declarative region.

Static Semantics

6 {*enumeration literal*} Each enumeration_literal_specification is the explicit declaration of the corresponding *enumeration literal*: it declares a parameterless function, whose defining name is the defining_identifier or defining_character_literal, and whose result type is the enumeration type.

6.a **Reason:** This rule defines the profile of the enumeration literal, which is used in the various types of conformance.

6.b **Ramification:** The parameterless function associated with an enumeration literal is fully defined by the enumeration_type_definition; a body is not permitted for it, and it never fails the Elaboration_Check when called.

7 Each enumeration literal corresponds to a distinct value of the enumeration type, and to a distinct position number. {*position number (of an enumeration value)* [partial]} The position number of the value of the first listed enumeration literal is zero; the position number of the value of each subsequent enumeration literal is one more than that of its predecessor in the list.

8 [The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.]

9 [{*overloaded (enumeration literal)* [partial]} If the same defining_identifier or defining_character_literal is specified in more than one enumeration_type_definition, the corresponding enumeration literals are said to be *overloaded*. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal has to be determinable from the context (see 8.6).]

Dynamic Semantics

10 {*elaboration (enumeration_type_definition)* [partial]} {*constrained (subtype)*} {*unconstrained (subtype)*} The elaboration of an enumeration_type_definition creates the enumeration type and its first subtype, which is constrained to the base range of the type.

10.a **Ramification:** The first subtype of a discrete type is always constrained, except in the case of a derived type whose parent subtype is Whatever'Base.

11 When called, the parameterless function associated with an enumeration literal returns the corresponding value of the enumeration type.

NOTES

12 24 If an enumeration literal occurs in a context that does not otherwise suffice to determine the type of the literal, then qualification by the name of the enumeration type is one way to resolve the ambiguity (see 4.7).

Examples

13 *Examples of enumeration types and subtypes:*

14
```
type Day    is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Suit   is (Clubs, Diamonds, Hearts, Spades);
type Gender is (M, F);
type Level  is (Low, Medium, Urgent);
type Color  is (White, Red, Yellow, Green, Blue, Brown, Black);
type Light  is (Red, Amber, Green); -- Red and Green are overloaded
```

15
```
type Hexa   is ('A', 'B', 'C', 'D', 'E', 'F');
type Mixed  is ('A', 'B', '*', B, None, '?', '%');
```

16
```
subtype Weekday is Day   range Mon .. Fri;
subtype Major   is Suit  range Hearts .. Spades;
subtype Rainbow is Color range Red .. Blue;  -- the Color Red, not the Light
```

Wording Changes from Ada 83

16.a The syntax rule for defining_character_literal is new. It is used for the defining occurrence of a character_literal, analogously to defining_identifier. Usage occurrences use the name or selector_name syntactic categories.

16.b We emphasize the fact that an enumeration literal denotes a function, which is called to produce a value.

3.5.2 Character Types

Static Semantics

1 {*character type*} An enumeration type is said to be a *character type* if at least one of its enumeration literals is a character_literal.

2/2 {*AI95-00285-01*} {*Latin-1*} {*BMP*} {*ISO/IEC 10646:2003*} {*Character*} The predefined type Character is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding character_literal in Character. Each of the nongraphic positions of Row 00 (0000-001F and 007F-009F) has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes Image, Wide_Image, Wide_Wide_Image, Value, Wide_Value, and Wide_Wide_Value; these names are given in the definition of type Character in A.1, "The Package Standard", but are set in *italics*. {*italics (nongraphic characters)*}

3/2 {*AI95-00285-01*} {*Wide_Character*} {*BMP*} {*ISO/IEC 10646:2003*} The predefined type Wide_Character is a character type whose values correspond to the 65536 code positions of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of the BMP has a corresponding character_literal in Wide_Character. The first 256 values of Wide_Character have the same character_literal or language-defined name as defined for Character. Each of the graphic_characters has a corresponding character_literal.

3.1/2 {*AI95-00285-01*} {*Wide_Wide_Character*} {*BMP*} {*ISO/IEC 10646:2003*} The predefined type Wide_Wide_Character is a character type whose values correspond to the 2147483648 code positions of the ISO/IEC 10646:2003 character set. Each of the graphic_characters has a corresponding character_literal in Wide_Wide_Character. The first 65536 values of Wide_Wide_Character have the same character_literal or language-defined name as defined for Wide_Character.

3.2/2 {*AI95-00285-01*} The characters whose code position is larger than 16#FF# and which are not graphic_characters have language-defined names which are formed by appending to the string "Hex_" the representation of their code position in hexadecimal as eight extended digits. As with other language-defined names, these names are usable only with the attributes (Wide_)Wide_Image and (Wide_)Wide_Value; they are not usable as enumeration literals.

3.a/2 **Reason:** {*AI95-00285-01*} The language-defined names are not usable as enumeration literals to avoid "polluting" the name space. Since Wide_Character and Wide_Wide_Character are defined in Standard, if the language-defined names were usable as enumeration literals, they would hide other nonoverloadable declarations with the same names in **use**-d packages.]}

Implementation Permissions

4/2 *This paragraph was deleted.*{*AI95-00285-01*}

Implementation Advice

5/2 *This paragraph was deleted.*{*AI95-00285-01*}

NOTES

6 25 The language-defined library package Characters.Latin_1 (see A.3.3) includes the declaration of constants denoting control characters, lower case characters, and special characters of the predefined type Character.

6.a **To be honest:** The package ASCII does the same, but only for the first 128 characters of Character. Hence, it is an obsolescent package, and we no longer mention it here.

7 26 A conventional character set such as *EBCDIC* can be declared as a character type; the internal codes of the characters can be specified by an enumeration_representation_clause as explained in clause 13.4.

Examples

8 *Example of a character type:*

9
```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
```

Inconsistencies With Ada 83

9.a {*inconsistencies with Ada 83*} The declaration of Wide_Character in package Standard hides use-visible declarations with the same defining identifier. In the unlikely event that an Ada 83 program had depended on such a use-visible declaration, and the program remains legal after the substitution of Standard.Wide_Character, the meaning of the program will be different.

Incompatibilities With Ada 83

9.b {*incompatibilities with Ada 83*} The presence of Wide_Character in package Standard means that an expression such as

9.c
```
'a' = 'b'
```

9.d is ambiguous in Ada 95, whereas in Ada 83 both literals could be resolved to be of type Character.

9.e The change in visibility rules (see 4.2) for character literals means that additional qualification might be necessary to resolve expressions involving overloaded subprograms and character literals.

Extensions to Ada 83

9.f {*extensions to Ada 83*} The type Character has been extended to have 256 positions, and the type Wide_Character has been added. Note that this change was already approved by the ARG for Ada 83 conforming compilers.

9.g The rules for referencing character literals are changed (see 4.2), so that the declaration of the character type need not be directly visible to use its literals, similar to **null** and string literals. Context is used to resolve their type.

Inconsistencies With Ada 95

9.h/2 {*AI95-00285-01*} {*inconsistencies with Ada 95*} Ada 95 defined most characters in Wide_Character to be graphic characters, while Ada 2005 uses the categorizations from ISO-10646:2003. It also provides language-defined names for all non-graphic characters. That means that in Ada 2005, Wide_Character'Wide_Value will raise Constraint_Error for a string representing a character_literal of a non-graphic character, while Ada 95 would have accepted it. Similarly, the result of Wide_Character'Wide_Image will change for such non-graphic characters.

9.i/2 {*AI95-00395-01*} The language-defined names FFFE and FFFF were replaced by a consistent set of language-defined names for all non-graphic characters with positions greater than 16#FF#. That means that in Ada 2005, Wide_Character'Wide_Value("FFFE") will raise Constraint_Error while Ada 95 would have accepted it. Similarly, the result of Wide_Character'Wide_Image will change for the position numbers 16#FFFE# and 16#FFFF#. It is very unlikely that this will matter in practice, as these names do not represent useable characters.

9.j/2 {*AI95-00285-01*} {*AI95-00395-01*} Because of the previously mentioned changes to the Wide_Character'Wide_Image of various character values, the value of attribute Wide_Width will change for some subtypes of Wide_Character. However, the new language-defined names were chosen so that the value of Wide_Character'Wide_Width itself does not change.

9.k/2 {*AI95-00285-01*} The declaration of Wide_Wide_Character in package Standard hides use-visible declarations with the same defining identifier. In the (very) unlikely event that an Ada 95 program had depended on such a use-visible declaration, and the program remains legal after the substitution of Standard.Wide_Wide_Character, the meaning of the program will be different.

Extensions to Ada 95

9.l/2 {*AI95-00285-01*} {*extensions to Ada 95*} The type Wide_Wide_Character is new.

Wording Changes from Ada 95

9.m/2 {*AI95-00285-01*} Characters are now defined in terms of the entire ISO/IEC 10646:2003 character set.

9.n/2 {*AI95-00285-01*} We dropped the Implementation Advice for non-standard interpretation of character sets; an implementation can do what it wants in a non-standard mode, so there isn't much point to any advice.

3.5.3 Boolean Types

Static Semantics

1 {*Boolean*} There is a predefined enumeration type named Boolean, [declared in the visible part of package Standard]. {*False*} {*True*} It has the two enumeration literals False and True ordered with the relation False < True. {*boolean type*} Any descendant of the predefined type Boolean is called a *boolean* type.

1.a **Implementation Note:** An implementation is not required to support enumeration representation clauses on boolean types that impose an unacceptable implementation burden. See 13.4, “Enumeration Representation Clauses”. However, it is generally straightforward to support representations where False is zero and True is 2**n – 1 for some n.

3.5.4 Integer Types

1 {*integer type*} {*signed integer type*} {*modular type*} An integer_type_definition defines an integer type; it defines either a *signed* integer type, or a *modular* integer type. The base range of a signed integer type includes at least the values of the specified range. A modular type is an integer type with all arithmetic modulo a specified positive *modulus*; such a type corresponds to an unsigned type with wrap-around semantics. {*unsigned type: See modular type*}

Syntax

2 integer_type_definition ::= signed_integer_type_definition | modular_type_definition

3 signed_integer_type_definition ::= **range** *static*_simple_expression .. *static*_simple_expression

3.a **Discussion:** We don't call this a range_constraint, because it is rather different — not only is it required to be static, but the associated overload resolution rules are different than for normal range constraints. A similar comment applies to real_range_specification. This used to be integer_range_specification but when we added support for modular types, it seemed overkill to have three levels of syntax rules, and just calling these signed_integer_range_specification and modular_range_specification loses the fact that they are defining different classes of types, which is important for the generic type matching rules.

4 modular_type_definition ::= **mod** *static*_expression

Name Resolution Rules

5 {*expected type (signed_integer_type_definition simple_expression)* [partial]} Each simple_expression in a signed_integer_type_definition is expected to be of any integer type; they need not be of the same type. {*expected type (modular_type_definition expression)* [partial]} The expression in a modular_type_definition is likewise expected to be of any integer type.

Legality Rules

6 The simple_expressions of a signed_integer_type_definition shall be static, and their values shall be in the range System.Min_Int .. System.Max_Int.

7 {*modulus (of a modular type)*} {*Max_Binary_Modulus*} {*Max_Nonbinary_Modulus*} The expression of a modular_type_definition shall be static, and its value (the *modulus*) shall be positive, and shall be no greater than System.Max_Binary_Modulus if a power of 2, or no greater than System.Max_Nonbinary_Modulus if not.

7.a **Reason:** For a 2's-complement machine, supporting nonbinary moduli greater than System.Max_Int can be quite difficult, whereas essentially any binary moduli are straightforward to support, up to 2*System.Max_Int+2, so this justifies having two separate limits.

Static Semantics

8 The set of values for a signed integer type is the (infinite) set of mathematical integers[, though only values of the base range of the type are fully supported for run-time operations]. The set of values for a modular integer type are the values from 0 to one less than the modulus, inclusive.

9 {*base range (of a signed integer type)* [partial]} A signed_integer_type_definition defines an integer type whose base range includes at least the values of the simple_expressions and is symmetric about zero, excepting possibly an extra negative value. {*constrained (subtype)*} {*unconstrained (subtype)*} A signed_integer_type_definition also defines a constrained first subtype of the type, with a range whose bounds are given by the values of the simple_expressions, converted to the type being defined.

9.a/2 **Implementation Note:** {*AI95-00114-01*} The base range of a signed integer type might be much larger than is necessary to satisfy the above requirements.

9.a.1/1 **To be honest:** The conversion mentioned above is not an *implicit subtype conversion* (which is something that happens at overload resolution, see 4.6), although it happens implicitly. Therefore, the freezing rules are not invoked on the type (which is important so that representation items can be given for the type). {*subtype conversion (bounds of signed integer type)* [partial]}

10 {*base range (of a modular type)* [partial]} A modular_type_definition defines a modular type whose base range is from zero to one less than the given modulus. {*constrained (subtype)*} {*unconstrained (subtype)*} A modular_type_definition also defines a constrained first subtype of the type with a range that is the same as the base range of the type.

11 {*Integer*} There is a predefined signed integer subtype named Integer[, declared in the visible part of package Standard]. It is constrained to the base range of its type.

11.a **Reason:** Integer is a constrained subtype, rather than an unconstrained subtype. This means that on assignment to an object of subtype Integer, a range check is required. On the other hand, an object of subtype Integer'Base is unconstrained, and no range check (only overflow check) is required on assignment. For example, if the object is held in an extended-length register, its value might be outside of Integer'First .. Integer'Last. All parameter and result subtypes of the predefined integer operators are of such unconstrained subtypes, allowing extended-length registers to be used as operands or for the result. In an earlier version of Ada 95, Integer was unconstrained. However, the fact that certain Constraint_Errors might be omitted or appear elsewhere was felt to be an undesirable upward inconsistency in this case. Note that for Float, the opposite conclusion was reached, partly because of the high cost of performing range checks when not actually necessary. Objects of subtype Float are unconstrained, and no range checks, only overflow checks, are performed for them.

12 {*Natural*} {*Positive*} Integer has two predefined subtypes, [declared in the visible part of package Standard:]

```
    subtype Natural  is Integer range 0 .. Integer'Last;                13
    subtype Positive is Integer range 1 .. Integer'Last;
```

{*root_integer*} {*Min_Int*} {*Max_Int*} A type defined by an integer_type_definition is implicitly derived from *root_integer*, an anonymous predefined (specific) integer type, whose base range is System.Min_Int .. System.Max_Int. However, the base range of the new type is not inherited from *root_integer*, but is instead determined by the range or modulus specified by the integer_type_definition. {*universal_integer* [partial]} {*integer literals*} [Integer literals are all of the type *universal_integer*, the universal type (see 3.4.1) for the class rooted at *root_integer*, allowing their use with the operations of any integer type.] 14

Discussion: This implicit derivation is not considered exactly equivalent to explicit derivation via a derived_type_definition. In particular, integer types defined via a derived_type_definition inherit their base range from their parent type. A type defined by an integer_type_definition does not necessarily inherit its base range from *root_integer*. It is not specified whether the implicit derivation from *root_integer* is direct or indirect, not that it really matters. All we want is for all integer types to be descendants of *root_integer*. 14.a

{*8652/0099*} {*AI95-00152-01*} Note that this derivation does not imply any inheritance of subprograms. Subprograms are inherited only for types derived by a derived_type_definition (see 3.4), or a private_extension_declaration (see 7.3, 7.3.1, and 12.5.1). 14.a.1/1

Implementation Note: It is the intent that even nonstandard integer types (see below) will be descendants of *root_integer*, even though they might have a base range that exceeds that of *root_integer*. This causes no problem for static calculations, which are performed without range restrictions (see 4.9). However for run-time calculations, it is possible that Constraint_Error might be raised when using an operator of *root_integer* on the result of 'Val applied to a value of a nonstandard integer type. 14.b

{*position number (of an integer value)* [partial]} The *position number* of an integer value is equal to the value. 15

{*AI95-00340-01*} For every modular subtype S, the following attributes are defined: 16/2

S'Mod {*AI95-00340-01*} S'Mod denotes a function with the following specification: 16.1/2

```
        function S'Mod (Arg : universal_integer)                    16.2/2
          return S'Base
```

This function returns *Arg* **mod** S'Modulus, as a value of the type of S. 16.3/2

S'Modulus S'Modulus yields the modulus of the type of S, as a value of the type *universal_integer*. 17

Dynamic Semantics

{*elaboration (integer_type_definition)* [partial]} The elaboration of an integer_type_definition creates the integer type and its first subtype. 18

For a modular type, if the result of the execution of a predefined operator (see 4.5) is outside the base range of the type, the result is reduced modulo the modulus of the type to a value that is within the base range of the type. 19

{*Overflow_Check* [partial]} {*check, language-defined (Overflow_Check)*} {*Constraint_Error (raised by failure of run-time check)*} For a signed integer type, the exception Constraint_Error is raised by the execution of an operation that cannot deliver the correct result because it is outside the base range of the type. [{*Division_Check* [partial]} {*check, language-defined (Division_Check)*} {*Constraint_Error (raised by failure of run-time check)*} For any integer type, Constraint_Error is raised by the operators "/", "**rem**", and "**mod**" if the right operand is zero.] 20

Implementation Requirements

{*Integer*} In an implementation, the range of Integer shall include the range –2**15+1 .. +2**15–1. 21

{*Long_Integer*} If Long_Integer is predefined for an implementation, then its range shall include the range –2**31+1 .. +2**31–1. 22

System.Max_Binary_Modulus shall be at least 2**16. 23

Implementation Permissions

24 For the execution of a predefined operation of a signed integer type, the implementation need not raise Constraint_Error if the result is outside the base range of the type, so long as the correct result is produced.

24.a **Discussion:** Constraint_Error is never raised for operations on modular types, except for divide-by-zero (and **rem**/**mod**-by-zero).

25 {*Long_Integer*} {*Short_Integer*} An implementation may provide additional predefined signed integer types[, declared in the visible part of Standard], whose first subtypes have names of the form Short_Integer, Long_Integer, Short_Short_Integer, Long_Long_Integer, etc. Different predefined integer types are allowed to have the same base range. However, the range of Integer should be no wider than that of Long_Integer. Similarly, the range of Short_Integer (if provided) should be no wider than Integer. Corresponding recommendations apply to any other predefined integer types. There need not be a named integer type corresponding to each distinct base range supported by an implementation. The range of each first subtype should be the base range of its type.

25.a **Implementation defined:** The predefined integer types declared in Standard.

26 {*nonstandard integer type*} An implementation may provide *nonstandard integer types*, descendants of *root_integer* that are declared outside of the specification of package Standard, which need not have all the standard characteristics of a type defined by an integer_type_definition. For example, a nonstandard integer type might have an asymmetric base range or it might not be allowed as an array or loop index (a very long integer). Any type descended from a nonstandard integer type is also nonstandard. An implementation may place arbitrary restrictions on the use of such types; it is implementation defined whether operators that are predefined for "any integer type" are defined for a particular nonstandard integer type. [In any case, such types are not permitted as explicit_generic_actual_parameters for formal scalar types — see 12.5.2.]

26.a **Implementation defined:** Any nonstandard integer types and the operators defined for them.

27 {*one's complement (modular types)* [partial]} For a one's complement machine, the high bound of the base range of a modular type whose modulus is one less than a power of 2 may be equal to the modulus, rather than one less than the modulus. It is implementation defined for which powers of 2, if any, this permission is exercised.

27.1/1 {*8652/0003*} {*AI95-00095-01*} For a one's complement machine, implementations may support non-binary modulus values greater than System.Max_Nonbinary_Modulus. It is implementation defined which specific values greater than System.Max_Nonbinary_Modulus, if any, are supported.

27.a.1/1 **Reason:** On a one's complement machine, the natural full word type would have a modulus of 2**Word_Size–1. However, we would want to allow the all-ones bit pattern (which represents negative zero as a number) in logical operations. These permissions are intended to allow that and the natural modulus value without burdening implementations with supporting expensive modulus values.

Implementation Advice

28 {*Long_Integer*} An implementation should support Long_Integer in addition to Integer if the target machine supports 32-bit (or longer) arithmetic. No other named integer subtypes are recommended for package Standard. Instead, appropriate named integer subtypes should be provided in the library package Interfaces (see B.2).

28.a.1/2 **Implementation Advice:** Long_Integer should be declared in Standard if the target supports 32-bit arithmetic. No other named integer subtypes should be declared in Standard.

28.a **Implementation Note:** To promote portability, implementations should explicitly declare the integer (sub)types Integer and Long_Integer in Standard, and leave other predefined integer types anonymous. For implementations that already support Byte_Integer, etc., upward compatibility argues for keeping such declarations in Standard during the transition period, but perhaps generating a warning on use. A separate package Interfaces in the predefined environment is available for pre-declaring types such as Integer_8, Integer_16, etc. See B.2. In any case, if the user declares a subtype (first or not) whose range fits in, for example, a byte, the implementation can store variables of the subtype in a single byte, even if the base range of the type is wider.

{*two's complement (modular types)* [partial]} An implementation for a two's complement machine should support modular types with a binary modulus up to System.Max_Int*2+2. An implementation should support a nonbinary modulus up to Integer'Last. 29

Implementation Advice: For a two's complement target, modular types with a binary modulus up to System.Max_Int*2+2 should be supported. A nonbinary modulus up to Integer'Last should be supported. 29.a.1/2

Reason: Modular types provide bit-wise "**and**", "**or**", "**xor**", and "**not**" operations. It is important for systems programming that these be available for all integer types of the target hardware. 29.a

Ramification: Note that on a one's complement machine, the largest supported modular type would normally have a nonbinary modulus. On a two's complement machine, the largest supported modular type would normally have a binary modulus. 29.b

Implementation Note: Supporting a nonbinary modulus greater than Integer'Last can impose an undesirable implementation burden on some machines. 29.c

NOTES

27 {*universal_integer*} {*integer literals*} Integer literals are of the anonymous predefined integer type *universal_integer*. Other integer types have no literals. However, the overload resolution rules (see 8.6, "The Context of Overload Resolution") allow expressions of the type *universal_integer* whenever an integer type is expected. 30

28 The same arithmetic operators are predefined for all signed integer types defined by a signed_integer_type_definition (see 4.5, "Operators and Expression Evaluation"). For modular types, these same operators are predefined, plus bit-wise logical operators (**and**, **or**, **xor**, and **not**). In addition, for the unsigned types declared in the language-defined package Interfaces (see B.2), functions are defined that provide bit-wise shifting and rotating. 31

29 Modular types match a generic_formal_parameter_declaration of the form "**type** T **is mod** <>;"; signed integer types match "**type** T **is range** <>;" (see 12.5.2). 32

Examples

Examples of integer types and subtypes: 33

34
```
type Page_Num  is range 1 .. 2_000;
type Line_Size is range 1 .. Max_Line_Size;
```

35
```
subtype Small_Int   is Integer   range -10 .. 10;
subtype Column_Ptr  is Line_Size range 1 .. 10;
subtype Buffer_Size is Integer   range 0 .. Max;
```

36
```
type Byte        is mod 256; -- an unsigned byte
type Hash_Index  is mod 97;  -- modulus is prime
```

Extensions to Ada 83

{*extensions to Ada 83*} An implementation is allowed to support any number of distinct base ranges for integer types, even if fewer integer types are explicitly declared in Standard. 36.a

Modular (unsigned, wrap-around) types are new. 36.b

Wording Changes from Ada 83

Ada 83's integer types are now called "signed" integer types, to contrast them with "modular" integer types. 36.c

Standard.Integer, Standard.Long_Integer, etc., denote constrained subtypes of predefined integer types, consistent with the Ada 95 model that only subtypes have names. 36.d

We now impose minimum requirements on the base range of Integer and Long_Integer. 36.e

We no longer explain integer type definition in terms of an equivalence to a normal type derivation, except to say that all integer types are by definition implicitly derived from *root_integer*. This is for various reasons. 36.f

First of all, the equivalence with a type derivation and a subtype declaration was not perfect, and was the source of various AIs (for example, is the conversion of the bounds static? Is a numeric type a derived type with respect to other rules of the language?) 36.g

Secondly, we don't want to require that every integer size supported shall have a corresponding named type in Standard. Adding named types to Standard creates nonportabilities. 36.h

36.i Thirdly, we don't want the set of types that match a formal derived type "type T is new Integer;" to depend on the particular underlying integer representation chosen to implement a given user-defined integer type. Hence, we would have needed anonymous integer types as parent types for the implicit derivation anyway. We have simply chosen to identify only one anonymous integer type — *root_integer*, and stated that every integer type is derived from it.

36.j Finally, the "fiction" that there were distinct preexisting predefined types for every supported representation breaks down for fixed point with arbitrary smalls, and was never exploited for enumeration types, array types, etc. Hence, there seems little benefit to pushing an explicit equivalence between integer type definition and normal type derivation.

Extensions to Ada 95

36.k/2 {*AI95-00340-01*} {*extensions to Ada 95*} The Mod attribute is new. It eases mixing of signed and unsigned values in an expression, which can be difficult as there may be no type which can contain all of the values of both of the types involved.

Wording Changes from Ada 95

36.l/2 {*8652/0003*} {*AI95-00095-01*} **Corrigendum:** Added additional permissions for modular types on one's complement machines.

3.5.5 Operations of Discrete Types

Static Semantics

1 For every discrete subtype S, the following attributes are defined:

2 S'Pos S'Pos denotes a function with the following specification:

3
```
function S'Pos(Arg : S'Base)
  return universal_integer
```

4 This function returns the position number of the value of *Arg*, as a value of type *universal_integer*.

5 S'Val S'Val denotes a function with the following specification:

6
```
function S'Val(Arg : universal_integer)
  return S'Base
```

7 {*evaluation (Val)* [partial]} {*Constraint_Error (raised by failure of run-time check)*} This function returns a value of the type of S whose position number equals the value of *Arg*. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} For the evaluation of a call on S'Val, if there is no value in the base range of its type with the given position number, Constraint_Error is raised.

7.a **Ramification:** By the overload resolution rules, a formal parameter of type *universal_integer* allows an actual parameter of any integer type.

7.b **Reason:** We considered allowing S'Val for a signed integer subtype S to return an out-of-range value, but since checks were required for enumeration and modular types anyway, the allowance didn't seem worth the complexity of the rule.

Implementation Advice

8 For the evaluation of a call on S'Pos for an enumeration subtype, if the value of the operand does not correspond to the internal code for any enumeration literal of its type [(perhaps due to an uninitialized variable)], then the implementation should raise Program_Error. {*Program_Error (raised by failure of run-time check)*} This is particularly important for enumeration types with noncontiguous internal codes specified by an enumeration_representation_clause.

8.a.1/2 **Implementation Advice:** Program_Error should be raised for the evaluation of S'Pos for an enumeration type, if the value of the operand does not correspond to the internal code for any enumeration literal of the type.

8.a **Reason:** We say Program_Error here, rather than Constraint_Error, because the main reason for such values is uninitialized variables, and the normal way to indicate such a use (if detected) is to raise Program_Error. (Other reasons would involve the misuse of low-level features such as Unchecked_Conversion.)

NOTES

9 30 Indexing and loop iteration use values of discrete types.

31 {*predefined operations (of a discrete type)* [partial]} The predefined operations of a discrete type include the assignment operation, qualification, the membership tests, and the relational operators; for a boolean type they include the short-circuit control forms and the logical operators; for an integer type they include type conversion to and from other numeric types, as well as the binary and unary adding operators – and +, the multiplying operators, the unary operator **abs**, and the exponentiation operator. The assignment operation is described in 5.2. The other predefined operations are described in Section 4. 10

32 As for all types, objects of a discrete type have Size and Address attributes (see 13.3). 11

33 For a subtype of a discrete type, the result delivered by the attribute Val might not belong to the subtype; similarly, the actual parameter of the attribute Pos need not belong to the subtype. The following relations are satisfied (in the absence of an exception) by these attributes: 12

13
```
S'Val(S'Pos(X)) = X
S'Pos(S'Val(N)) = N
```

Examples

Examples of attributes of discrete subtypes: 14

-- *For the types and subtypes declared in subclause 3.5.1 the following hold:* 15

16
```
--  Color'First   = White,   Color'Last   = Black
--  Rainbow'First = Red,     Rainbow'Last = Blue
```

17
```
--  Color'Succ(Blue) = Rainbow'Succ(Blue) = Brown
--  Color'Pos(Blue)  = Rainbow'Pos(Blue)  = 4
--  Color'Val(0)     = Rainbow'Val(0)     = White
```

Extensions to Ada 83

{*extensions to Ada 83*} The attributes S'Succ, S'Pred, S'Width, S'Image, and S'Value have been generalized to apply to real types as well (see 3.5, “Scalar Types”). 17.a

3.5.6 Real Types

{*real type*} Real types provide approximations to the real numbers, with relative bounds on errors for floating point types, and with absolute bounds for fixed point types. 1

Syntax

real_type_definition ::=
  floating_point_definition | fixed_point_definition 2

Static Semantics

{*root_real*} A type defined by a real_type_definition is implicitly derived from *root_real*, an anonymous predefined (specific) real type. [Hence, all real types, whether floating point or fixed point, are in the derivation class rooted at *root_real*.] 3

Ramification: It is not specified whether the derivation from *root_real* is direct or indirect, not that it really matters. All we want is for all real types to be descendants of *root_real*. 3.a

{*8652/0099*} {*AI95-00152-01*} Note that this derivation does not imply any inheritance of subprograms. Subprograms are inherited only for types derived by a derived_type_definition (see 3.4), or a private_extension_declaration (see 7.3, 7.3.1, and 12.5.1). 3.a.1/1

[{*universal_real* [partial]} {*real literals*} Real literals are all of the type *universal_real*, the universal type (see 3.4.1) for the class rooted at *root_real*, allowing their use with the operations of any real type. {*universal_fixed* [partial]} Certain multiplying operators have a result type of *universal_fixed* (see 4.5.5), the universal type for the class of fixed point types, allowing the result of the multiplication or division to be used where any specific fixed point type is expected.] 4

Dynamic Semantics

5 {*elaboration (real_type_definition)* [partial]} The elaboration of a real_type_definition consists of the elaboration of the floating_point_definition or the fixed_point_definition.

Implementation Requirements

6 An implementation shall perform the run-time evaluation of a use of a predefined operator of *root_real* with an accuracy at least as great as that of any floating point type definable by a floating_point_definition.

6.a **Ramification:** Static calculations using the operators of *root_real* are exact, as for all static calculations. See 4.9.

6.b **Implementation Note:** The Digits attribute of the type used to represent *root_real* at run time is at least as great as that of any other floating point type defined by a floating_point_definition, and its safe range includes that of any such floating point type with the same Digits attribute. On some machines, there might be real types with less accuracy but a wider range, and hence run-time calculations with *root_real* might not be able to accommodate all values that can be represented at run time in such floating point or fixed point types.

Implementation Permissions

7/2 {*AI95-00114-01*} [For the execution of a predefined operation of a real type, the implementation need not raise Constraint_Error if the result is outside the base range of the type, so long as the correct result is produced, or the Machine_Overflows attribute of the type is False (see G.2).]

8 {*nonstandard real type*} An implementation may provide *nonstandard real types*, descendants of *root_real* that are declared outside of the specification of package Standard, which need not have all the standard characteristics of a type defined by a real_type_definition. For example, a nonstandard real type might have an asymmetric or unsigned base range, or its predefined operations might wrap around or “saturate” rather than overflow (modular or saturating arithmetic), or it might not conform to the accuracy model (see G.2). Any type descended from a nonstandard real type is also nonstandard. An implementation may place arbitrary restrictions on the use of such types; it is implementation defined whether operators that are predefined for “any real type” are defined for a particular nonstandard real type. [In any case, such types are not permitted as explicit_generic_actual_parameters for formal scalar types — see 12.5.2.]

8.a **Implementation defined:** Any nonstandard real types and the operators defined for them.

NOTES

9 34 As stated, real literals are of the anonymous predefined real type *universal_real*. Other real types have no literals. However, the overload resolution rules (see 8.6) allow expressions of the type *universal_real* whenever a real type is expected.

Wording Changes from Ada 83

9.a The syntax rule for real_type_definition is modified to use the new syntactic categories floating_point_definition and fixed_point_definition, instead of floating_point_constraint and fixed_point_constraint, because the semantics of a type definition are significantly different than the semantics of a constraint.

9.b All discussion of model numbers, safe ranges, and machine numbers is moved to 3.5.7, 3.5.8, and G.2. Values of a fixed point type are now described as being multiples of the *small* of the fixed point type, and we have no need for model numbers, safe ranges, etc. for fixed point types.

3.5.7 Floating Point Types

1 {*floating point type*} For floating point types, the error bound is specified as a relative precision by giving the required minimum number of significant decimal digits.

Syntax

2 floating_point_definition ::=
  digits *static*_expression [real_range_specification]

real_range_specification ::=
  range *static*_simple_expression .. *static*_simple_expression 3

Name Resolution Rules

{*requested decimal precision (of a floating point type)*} The *requested decimal precision*, which is the minimum number of significant decimal digits required for the floating point type, is specified by the value of the expression given after the reserved word **digits**. {*expected type (requested decimal precision)* [partial]} This expression is expected to be of any integer type. 4

{*expected type (real_range_specification bounds)* [partial]} Each simple_expression of a real_range_specification is expected to be of any real type[; the types need not be the same]. 5

Legality Rules

{*Max_Base_Digits*} The requested decimal precision shall be specified by a static expression whose value is positive and no greater than System.Max_Base_Digits. Each simple_expression of a real_range_specification shall also be static. {*Max_Digits*} If the real_range_specification is omitted, the requested decimal precision shall be no greater than System.Max_Digits. 6

Reason: We have added Max_Base_Digits to package System. It corresponds to the requested decimal precision of *root_real*. System.Max_Digits corresponds to the maximum value for Digits that may be specified in the absence of a real_range_specification, for upward compatibility. These might not be the same if *root_real* has a base range that does not include ± 10.0**(4*Max_Base_Digits). 6.a

A floating_point_definition is illegal if the implementation does not support a floating point type that satisfies the requested decimal precision and range. 7

Implementation defined: What combinations of requested decimal precision and range are supported for floating point types. 7.a

Static Semantics

The set of values for a floating point type is the (infinite) set of rational numbers. {*machine numbers (of a floating point type)*} The *machine numbers* of a floating point type are the values of the type that can be represented exactly in every unconstrained variable of the type. {*base range (of a floating point type)* [partial]} The base range (see 3.5) of a floating point type is symmetric around zero, except that it can include some extra negative values in some implementations. 8

Implementation Note: For example, if a 2's complement representation is used for the mantissa rather than a sign-mantissa or 1's complement representation, then there is usually one extra negative machine number. 8.a

To be honest: If the Signed_Zeros attribute is True, then minus zero could in a sense be considered a value of the type. However, for most purposes, minus zero behaves the same as plus zero. 8.b

{*base decimal precision (of a floating point type)*} The *base decimal precision* of a floating point type is the number of decimal digits of precision representable in objects of the type. {*safe range (of a floating point type)*} The *safe range* of a floating point type is that part of its base range for which the accuracy corresponding to the base decimal precision is preserved by all predefined operations. 9

Implementation Note: In most cases, the safe range and base range are the same. However, for some hardware, values near the boundaries of the base range might result in excessive inaccuracies or spurious overflows when used with certain predefined operations. For such hardware, the safe range would omit such values. 9.a

{*base decimal precision (of a floating point type)* [partial]} A floating_point_definition defines a floating point type whose base decimal precision is no less than the requested decimal precision. {*safe range (of a floating point type)* [partial]} {*base range (of a floating point type)* [partial]} If a real_range_specification is given, the safe range of the floating point type (and hence, also its base range) includes at least the values of the simple expressions given in the real_range_specification. If a real_range_specification is not given, the safe (and base) range of the type includes at least the values of the range –10.0**(4*D) .. +10.0**(4*D) 10 where D is the requested decimal precision. [The safe range might include other values as well. The attributes Safe_First and Safe_Last give the actual bounds of the safe range.]

11 A floating_point_definition also defines a first subtype of the type. {*constrained (subtype)*} {*unconstrained (subtype)*} If a real_range_specification is given, then the subtype is constrained to a range whose bounds are given by a conversion of the values of the simple_expressions of the real_range_specification to the type being defined. Otherwise, the subtype is unconstrained.

11.a.1/1 **To be honest:** The conversion mentioned above is not an *implicit subtype conversion* (which is something that happens at overload resolution, see 4.6), although it happens implicitly. Therefore, the freezing rules are not invoked on the type (which is important so that representation items can be given for the type). {*subtype conversion (bounds of a floating point type)* [partial]}

12 {*Float*} There is a predefined, unconstrained, floating point subtype named Float[, declared in the visible part of package Standard].

Dynamic Semantics

13 {*elaboration (floating_point_definition)* [partial]} [The elaboration of a floating_point_definition creates the floating point type and its first subtype.]

Implementation Requirements

14 {*Float*} In an implementation that supports floating point types with 6 or more digits of precision, the requested decimal precision for Float shall be at least 6.

15 {*Long_Float*} If Long_Float is predefined for an implementation, then its requested decimal precision shall be at least 11.

Implementation Permissions

16 {*Short_Float*} {*Long_Float*} An implementation is allowed to provide additional predefined floating point types[, declared in the visible part of Standard], whose (unconstrained) first subtypes have names of the form Short_Float, Long_Float, Short_Short_Float, Long_Long_Float, etc. Different predefined floating point types are allowed to have the same base decimal precision. However, the precision of Float should be no greater than that of Long_Float. Similarly, the precision of Short_Float (if provided) should be no greater than Float. Corresponding recommendations apply to any other predefined floating point types. There need not be a named floating point type corresponding to each distinct base decimal precision supported by an implementation.

16.a **Implementation defined:** The predefined floating point types declared in Standard.

Implementation Advice

17 {*Long_Float*} An implementation should support Long_Float in addition to Float if the target machine supports 11 or more digits of precision. No other named floating point subtypes are recommended for package Standard. Instead, appropriate named floating point subtypes should be provided in the library package Interfaces (see B.2).

17.a.1/2 **Implementation Advice:** Long_Float should be declared in Standard if the target supports 11 or more digits of precision. No other named float subtypes should be declared in Standard.

17.a **Implementation Note:** To promote portability, implementations should explicitly declare the floating point (sub)types Float and Long_Float in Standard, and leave other predefined float types anonymous. For implementations that already support Short_Float, etc., upward compatibility argues for keeping such declarations in Standard during the transition period, but perhaps generating a warning on use. A separate package Interfaces in the predefined environment is available for pre-declaring types such as Float_32, IEEE_Float_64, etc. See B.2.

NOTES

18 35 If a floating point subtype is unconstrained, then assignments to variables of the subtype involve only Overflow_Checks, never Range_Checks.

Examples

Examples of floating point types and subtypes: 19

```
type Coefficient is digits 10 range -1.0 .. 1.0;                    20

type Real is digits 8;                                              21
type Mass is digits 7 range 0.0 .. 1.0E35;

subtype Probability is Real range 0.0 .. 1.0;   --  a subtype with a smaller range   22
```

Inconsistencies With Ada 83

22.a {*inconsistencies with Ada 83*} No Range_Checks, only Overflow_Checks, are performed on variables (or parameters) of an unconstrained floating point subtype. This is upward compatible for programs that do not raise Constraint_Error. For those that do raise Constraint_Error, it is possible that the exception will be raised at a later point, or not at all, if extended range floating point registers are used to hold the value of the variable (or parameter).

22.b **Reason:** This change was felt to be justified by the possibility of improved performance on machines with extended-range floating point registers. An implementation need not take advantage of this relaxation in the range checking; it can hide completely the use of extended range registers if desired, presumably at some run-time expense.

Wording Changes from Ada 83

22.c The syntax rules for floating_point_constraint and floating_accuracy_definition are removed. The syntax rules for floating_point_definition and real_range_specification are new.

22.d A syntax rule for digits_constraint is given in 3.5.9, “Fixed Point Types”. In J.3 we indicate that a digits_constraint may be applied to a floating point subtype_mark as well (to be compatible with Ada 83's floating_point_constraint).

22.e Discussion of model numbers is postponed to 3.5.8 and G.2. The concept of safe numbers has been replaced by the concept of the safe range of values. The bounds of the safe range are given by T'Safe_First .. T'Safe_Last, rather than -T'Safe_Large .. T'Safe_Large, since on some machines the safe range is not perfectly symmetric. The concept of machine numbers is new, and is relevant to the definition of Succ and Pred for floating point numbers.

3.5.8 Operations of Floating Point Types

Static Semantics

1 The following attribute is defined for every floating point subtype S:

2/1 S'Digits {*8652/0004*} {*AI95-00203-01*} S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. The requested decimal precision of the base subtype of a floating point type *T* is defined to be the largest value of *d* for which
ceiling(d * log(10) / log(T'Machine_Radix)) + g <= T'Model_Mantissa
where g is 0 if Machine_Radix is a positive power of 10 and 1 otherwise.

NOTES

3 36 {*predefined operations (of a floating point type)* [partial]} The predefined operations of a floating point type include the assignment operation, qualification, the membership tests, and explicit conversion to and from other numeric types. They also include the relational operators and the following predefined arithmetic operators: the binary and unary adding operators – and +, certain multiplying operators, the unary operator **abs**, and the exponentiation operator.

4 37 As for all types, objects of a floating point type have Size and Address attributes (see 13.3). Other attributes of floating point types are defined in A.5.3.

Wording Changes from Ada 95

4.a/2 {*8652/0004*} {*AI95-00203-01*} **Corrigendum:** Corrected the formula for Digits when the Machine_Radix is 10.

3.5.9 Fixed Point Types

1 {*fixed point type*} {*ordinary fixed point type*} {*decimal fixed point type*} A fixed point type is either an ordinary fixed point type, or a decimal fixed point type. {*delta (of a fixed point type)*} The error bound of a fixed point type is specified as an absolute value, called the *delta* of the fixed point type.

Syntax

2 fixed_point_definition ::= ordinary_fixed_point_definition | decimal_fixed_point_definition

3 ordinary_fixed_point_definition ::=
delta *static*_expression real_range_specification

4 decimal_fixed_point_definition ::=
delta *static*_expression **digits** *static*_expression [real_range_specification]

5 digits_constraint ::=
digits *static*_expression [range_constraint]

Name Resolution Rules

6 {*expected type (fixed point type delta)* [partial]} For a type defined by a fixed_point_definition, the *delta* of the type is specified by the value of the expression given after the reserved word **delta**; this expression is expected to be of any real type. {*expected type (decimal fixed point type digits)* [partial]} {*digits (of a decimal fixed point subtype)*} {*decimal fixed point type*} For a type defined by a decimal_fixed_point_definition (a *decimal* fixed point type), the number of significant decimal digits for its first subtype (the *digits* of the first subtype) is specified by the expression given after the reserved word **digits**; this expression is expected to be of any integer type.

Legality Rules

7 In a fixed_point_definition or digits_constraint, the expressions given after the reserved words **delta** and **digits** shall be static; their values shall be positive.

8/2 {*AI95-00100-01*} {*small (of a fixed point type)*} The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type.{*machine numbers (of a fixed point type)* [partial]} The *machine numbers* of a fixed point type are the values of the type that can be represented exactly in every unconstrained variable of the type. {*ordinary fixed point type*} For a type defined by an ordinary_fixed_point_definition (an *ordinary* fixed point type), the *small* may be specified by an attribute_definition_clause (see 13.3); if so specified, it shall be no greater than the *delta* of the type. If not specified, the *small* of an ordinary fixed point type is an implementation-defined power of two less than or equal to the *delta*.

8.a **Implementation defined:** The *small* of an ordinary fixed point type.

9 For a decimal fixed point type, the *small* equals the *delta*; the *delta* shall be a power of 10. If a real_range_specification is given, both bounds of the range shall be in the range –(10***digits*–1)**delta* .. +(10***digits*–1)**delta*.

10 A fixed_point_definition is illegal if the implementation does not support a fixed point type with the given *small* and specified range or *digits*.

10.a **Implementation defined:** What combinations of *small*, range, and *digits* are supported for fixed point types.

11 For a subtype_indication with a digits_constraint, the subtype_mark shall denote a decimal fixed point subtype.

11.a **To be honest:** Or, as an obsolescent feature, a floating point subtype is permitted — see J.3.

Static Semantics

12 {*base range (of a fixed point type)* [partial]} The base range (see 3.5) of a fixed point type is symmetric around zero, except possibly for an extra negative value in some implementations.

13 {*base range (of an ordinary fixed point type)* [partial]} An ordinary_fixed_point_definition defines an ordinary fixed point type whose base range includes at least all multiples of *small* that are between the bounds specified in the real_range_specification. The base range of the type does not necessarily include

the specified bounds themselves. {*constrained (subtype)*} {*unconstrained (subtype)*} An ordinary_fixed_point_definition also defines a constrained first subtype of the type, with each bound of its range given by the closer to zero of:

- the value of the conversion to the fixed point type of the corresponding expression of the real_range_specification; {*implicit subtype conversion (bounds of a fixed point type)* [partial]} 14

  To be honest: The conversion mentioned above is not an *implicit subtype conversion* (which is something that happens at overload resolution, see 4.6), although it happens implicitly. Therefore, the freezing rules are not invoked on the type (which is important so that representation items can be given for the type). {*subtype conversion (bounds of a fixed point type)* [partial]} 14.a.1/1

- the corresponding bound of the base range. 15

16 {*base range (of a decimal fixed point type)* [partial]} A decimal_fixed_point_definition defines a decimal fixed point type whose base range includes at least the range –(10***digits*–1)**delta* .. +(10***digits*–1)**delta*. {*constrained (subtype)*} {*unconstrained (subtype)*} A decimal_fixed_point_definition also defines a constrained first subtype of the type. If a real_range_specification is given, the bounds of the first subtype are given by a conversion of the values of the expressions of the real_range_specification. {*implicit subtype conversion (bounds of a decimal fixed point type)* [partial]} Otherwise, the range of the first subtype is –(10***digits*–1)**delta* .. +(10***digits*–1)**delta*.

16.a.1/1 **To be honest:** The conversion mentioned above is not an *implicit subtype conversion* (which is something that happens at overload resolution, see 4.6), although it happens implicitly. Therefore, the freezing rules are not invoked on the type (which is important so that representation items can be given for the type). {*subtype conversion (bounds of a decimal fixed point type)* [partial]}

Dynamic Semantics

17 {*elaboration (fixed_point_definition)* [partial]} The elaboration of a fixed_point_definition creates the fixed point type and its first subtype.

18 For a digits_constraint on a decimal fixed point subtype with a given *delta*, if it does not have a range_constraint, then it specifies an implicit range –(10***D*–1)**delta* .. +(10***D*–1)**delta*, where *D* is the value of the expression. {*compatibility (digits_constraint with a decimal fixed point subtype)*} A digits_constraint is *compatible* with a decimal fixed point subtype if the value of the expression is no greater than the *digits* of the subtype, and if it specifies (explicitly or implicitly) a range that is compatible with the subtype.

18.a **Discussion:** Except for the requirement that the *digits* specified be no greater than the *digits* of the subtype being constrained, a digits_constraint is essentially equivalent to a range_constraint.

18.b Consider the following example:

18.c
```
    type D is delta 0.01 digits 7 range -0.00 .. 9999.99;
```

18.d/1 The compatibility rule implies that the digits_constraint "**digits** 6" specifies an implicit range of "–9999.99 .. 9999.99". Thus, "**digits** 6" is not compatible with the constraint of D, but "**digits** 6 range 0.00 .. 9999.99" is compatible.

18.e/2 {*AI95-00114-01*} A value of a scalar type belongs to a constrained subtype of the type if it belongs to the range of the subtype. Attributes like Digits and Delta have no effect on this fundamental rule. So the obsolescent forms of digits_constraints and delta_constraints that are called “accuracy constraints” in RM83 don't really represent constraints on the values of the subtype, but rather primarily affect compatibility of the “constraint” with the subtype being “constrained.” In this sense, they might better be called “subtype assertions” rather than “constraints.”

18.f Note that the digits_constraint on a decimal fixed point subtype is a combination of an assertion about the *digits* of the subtype being further constrained, and a constraint on the range of the subtype being defined, either explicit or implicit.

19 {*elaboration (digits_constraint)* [partial]} The elaboration of a digits_constraint consists of the elaboration of the range_constraint, if any. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} If a range_constraint is given, a check is made that the bounds of the range are both in the range –(10***D*–1)**delta* .. +(10***D*–1)**delta*, where *D* is the value of the (static) expression given after the reserved word **digits**. {*Constraint_Error (raised by failure of run-time check)*} If this check fails, Constraint_Error is raised.

Implementation Requirements

20 The implementation shall support at least 24 bits of precision (including the sign bit) for fixed point types.

20.a **Reason:** This is sufficient to represent Standard.Duration with a *small* no more than 50 milliseconds.

Implementation Permissions

21 Implementations are permitted to support only *small*s that are a power of two. In particular, all decimal fixed point type declarations can be disallowed. Note however that conformance with the Information Systems Annex requires support for decimal *small*s, and decimal fixed point type declarations with *digits* up to at least 18.

21.a **Implementation Note:** The accuracy requirements for multiplication, division, and conversion (see G.2.1, "Model of Floating Point Arithmetic") are such that support for arbitrary *small*s should be practical without undue implementation effort. Therefore, implementations should support fixed point types with arbitrary values for *small* (within reason). One reasonable limitation would be to limit support to fixed point types that can be converted to the most precise floating point type without loss of precision (so that Fixed_IO is implementable in terms of Float_IO).

NOTES

22 38 The base range of an ordinary fixed point type need not include the specified bounds themselves so that the range specification can be given in a natural way, such as:

23
```
type Fraction is delta 2.0**(-15) range -1.0 .. 1.0;
```

24 With 2's complement hardware, such a type could have a signed 16-bit representation, using 1 bit for the sign and 15 bits for fraction, resulting in a base range of –1.0 .. 1.0–2.0**(–15).

Examples

25 *Examples of fixed point types and subtypes:*

26
```
type Volt is delta 0.125 range 0.0 .. 255.0;
```

27
```
   -- A pure fraction which requires all the available
   -- space in a word can be declared as the type Fraction:
type Fraction is delta System.Fine_Delta range -1.0 .. 1.0;
   -- Fraction'Last = 1.0 – System.Fine_Delta
```

28
```
type Money is delta 0.01 digits 15;  -- decimal fixed point
subtype Salary is Money digits 10;
   -- Money'Last = 10.0**13 – 0.01, Salary'Last = 10.0**8 – 0.01
```

Inconsistencies With Ada 83

28.a {*inconsistencies with Ada 83*} In Ada 95, S'Small always equals S'Base'Small, so if an implementation chooses a *small* for a fixed point type smaller than required by the *delta*, the value of S'Small in Ada 95 might not be the same as it was in Ada 83.

Extensions to Ada 83

28.b {*extensions to Ada 83*} Decimal fixed point types are new, though their capabilities are essentially similar to that available in Ada 83 with a fixed point type whose *small* equals its *delta* equals a power of 10. However, in the Information Systems Annex, additional requirements are placed on the support of decimal fixed point types (e.g. a minimum of 18 digits of precision).

Wording Changes from Ada 83

28.c The syntax rules for fixed_point_constraint and fixed_accuracy_definition are removed. The syntax rule for fixed_point_definition is new. A syntax rule for delta_constraint is included in the Obsolescent features (to be compatible with Ada 83's fixed_point_constraint).

Wording Changes from Ada 95

28.d/2 {*AI95-00100-01*} Added wording to define the machine numbers of fixed point types; this is needed by the static evaluation rules.

3.5.10 Operations of Fixed Point Types

Static Semantics

The following attributes are defined for every fixed point subtype S: 1

S'Small {*8652/0005*} {*AI95-00054-01*} S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. {*specifiable (of Small for fixed point types)* [partial]} {*Small clause*} Small may be specified for nonderived ordinary fixed point types via an attribute_definition_clause (see 13.3); the expression of such a clause shall be static. 2/1

S'Delta S'Delta denotes the *delta* of the fixed point subtype S. The value of this attribute is of the type *universal_real*. 3

Reason: The *delta* is associated with the *sub*type as opposed to the type, because of the possibility of an (obsolescent) delta_constraint. 3.a

S'Fore S'Fore yields the minimum number of characters needed before the decimal point for the decimal representation of any value of the subtype S, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute is of the type *universal_integer*. 4

S'Aft S'Aft yields the number of decimal digits needed after the decimal point to accommodate the *delta* of the subtype S, unless the *delta* of the subtype S is greater than 0.1, in which case the attribute yields the value one. [(S'Aft is the smallest positive integer N for which (10**N)*S'Delta is greater than or equal to one.)] The value of this attribute is of the type *universal_integer*. 5

The following additional attributes are defined for every decimal fixed point subtype S: 6

S'Digits S'Digits denotes the *digits* of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type *universal_integer*. Its value is determined as follows: {*digits (of a decimal fixed point subtype)*} 7

- For a first subtype or a subtype defined by a subtype_indication with a digits_constraint, the digits is the value of the expression given after the reserved word **digits**; 8
- For a subtype defined by a subtype_indication without a digits_constraint, the digits of the subtype is the same as that of the subtype denoted by the subtype_mark in the subtype_indication. 9

Implementation Note: Although a decimal subtype can be both range-constrained and digits-constrained, the digits constraint is intended to control the Size attribute of the subtype. For decimal types, Size can be important because input/output of decimal types is so common. 9.a

- The digits of a base subtype is the largest integer *D* such that the range –(10***D*–1)**delta* .. +(10***D*–1)**delta* is included in the base range of the type. 10

S'Scale S'Scale denotes the *scale* of the subtype S, defined as the value N such that S'Delta = 10.0**(–N). {*scale (of a decimal fixed point subtype)*} [The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S.] The value of this attribute is of the type *universal_integer*. 11

Ramification: S'Scale is negative if S'Delta is greater than one. By contrast, S'Aft is always positive. 11.a

S'Round S'Round denotes a function with the following specification: 12

13
```
function S'Round(X : universal_real)
  return S'Base
```

14 The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S).

NOTES

15 39 All subtypes of a fixed point type will have the same value for the Delta attribute, in the absence of delta_constraints (see J.3).

16 40 S'Scale is not always the same as S'Aft for a decimal subtype; for example, if S'Delta = 1.0 then S'Aft is 1 while S'Scale is 0.

17 41 {*predefined operations (of a fixed point type)* [partial]} The predefined operations of a fixed point type include the assignment operation, qualification, the membership tests, and explicit conversion to and from other numeric types. They also include the relational operators and the following predefined arithmetic operators: the binary and unary adding operators – and +, multiplying operators, and the unary operator **abs**.

18 42 As for all types, objects of a fixed point type have Size and Address attributes (see 13.3). Other attributes of fixed point types are defined in A.5.4.

Wording Changes from Ada 95

18.a/2 {*8652/0005*} {*AI95-00054-01*} **Corrigendum:** Clarified that *small* may be specified only for ordinary fixed point types.

3.6 Array Types

1 {*array*} {*array type*} An *array* object is a composite object consisting of components which all have the same subtype. The name for a component of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of the components.

Syntax

2 array_type_definition ::=
  unconstrained_array_definition | constrained_array_definition

3 unconstrained_array_definition ::=
  array(index_subtype_definition {, index_subtype_definition}) **of** component_definition

4 index_subtype_definition ::= subtype_mark **range** <>

5 constrained_array_definition ::=
  array (discrete_subtype_definition {, discrete_subtype_definition}) **of** component_definition

6 discrete_subtype_definition ::= *discrete*_subtype_indication | range

7/2 {*AI95-00230-01*} {*AI95-00406-01*} component_definition ::=
  [**aliased**] subtype_indication
 | [**aliased**] access_definition

Name Resolution Rules

8 {*expected type (discrete_subtype_definition range)* [partial]} For a discrete_subtype_definition that is a range, the range shall resolve to be of some specific discrete type[; which discrete type shall be determined without using any context other than the bounds of the range itself (plus the preference for *root_integer* — see 8.6).]

Legality Rules

9 {*index subtype*} Each index_subtype_definition or discrete_subtype_definition in an array_type_definition defines an *index subtype*; {*index type*} its type (the *index type*) shall be discrete.

9.a **Discussion:** {*index (of an array)*} An *index* is a discrete quantity used to select along a given dimension of an array. A component is selected by specifying corresponding values for each of the indices.

{*component subtype*} The subtype defined by the subtype_indication of a component_definition (the *component subtype*) shall be a definite subtype. 10

> **Ramification:** This applies to all uses of component_definition, including in record_type_definitions and protected_definitions. 10.a

This paragraph was deleted.{*AI95-00363-01*} 11/2

Static Semantics

{*dimensionality (of an array)*} {*one-dimensional array*} {*multi-dimensional array*} An array is characterized by the number of indices (the *dimensionality* of the array), the type and position of each index, the lower and upper bounds for each index, and the subtype of the components. The order of the indices is significant. 12

A one-dimensional array has a distinct component for each possible index value. A multidimensional array has a distinct component for each possible sequence of index values that can be formed by selecting one value for each index position (in the given order). The possible values for a given index are all the values between the lower and upper bounds, inclusive; {*index range*} this range of values is called the *index range*. {*bounds (of an array)*} The *bounds* of an array are the bounds of its index ranges. {*length (of a dimension of an array)*} The *length* of a dimension of an array is the number of values of the index range of the dimension (zero for a null range). {*length (of a one-dimensional array)*} The *length* of a one-dimensional array is the length of its only dimension. 13

An array_type_definition defines an array type and its first subtype. For each object of this array type, the number of indices, the type and position of each index, and the subtype of the components are as in the type definition[; the values of the lower and upper bounds for each index belong to the corresponding index subtype of its type, except for null arrays (see 3.6.1)]. 14

{*constrained (subtype)*} {*unconstrained (subtype)*} An unconstrained_array_definition defines an array type with an unconstrained first subtype. Each index_subtype_definition defines the corresponding index subtype to be the subtype denoted by the subtype_mark. [{*box (compound delimiter)* [partial]} The compound delimiter <> (called a *box*) of an index_subtype_definition stands for an undefined range (different objects of the type need not have the same bounds).] 15

{*constrained (subtype)*} {*unconstrained (subtype)*} A constrained_array_definition defines an array type with a constrained first subtype. Each discrete_subtype_definition defines the corresponding index subtype, as well as the corresponding index range for the constrained first subtype. {*constraint (of a first array subtype)* [partial]} The *constraint* of the first subtype consists of the bounds of the index ranges. 16

> **Discussion:** Although there is no namable unconstrained array subtype in this case, the predefined slicing and concatenation operations can operate on and yield values that do not necessarily belong to the first array subtype. This is also true for Ada 83. 16.a

The discrete subtype defined by a discrete_subtype_definition is either that defined by the subtype_-indication, or a subtype determined by the range as follows: 17

- If the type of the range resolves to *root_integer*, then the discrete_subtype_definition defines a subtype of the predefined type Integer with bounds given by a conversion to Integer of the bounds of the range; {*implicit subtype conversion (bounds of a range)* [partial]} 18

> **Reason:** This ensures that indexing over the discrete subtype can be performed with regular Integers, rather than only *universal_integer*s. 18.a

> **Discussion:** We considered doing this by simply creating a “preference” for Integer when resolving the range. {*Beaujolais effect* [partial]} However, this can introduce *Beaujolais* effects when the simple_expressions involve calls on functions visible due to **use** clauses. 18.b

- Otherwise, the discrete_subtype_definition defines a subtype of the type of the range, with the bounds given by the range. 19

20 {*nominal subtype (of a component)* [partial]} The component_definition of an array_type_definition defines the nominal subtype of the components. If the reserved word **aliased** appears in the component_definition, then each component of the array is aliased (see 3.10).

Dynamic Semantics

21 {*elaboration (array_type_definition)* [partial]} The elaboration of an array_type_definition creates the array type and its first subtype, and consists of the elaboration of any discrete_subtype_definitions and the component_definition.

22/2 {*8652/0002*} {*AI95-00171-01*} {*AI95-00230-01*} {*elaboration (discrete_subtype_definition)* [partial]} The elaboration of a discrete_subtype_definition that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the subtype_indication or the evaluation of the range. The elaboration of a discrete_subtype_definition that contains one or more per-object expressions is defined in 3.8. {*elaboration (component_definition)* [partial]} The elaboration of a component_definition in an array_type_definition consists of the elaboration of the subtype_indication or access_definition. The elaboration of any discrete_subtype_definitions and the elaboration of the component_definition are performed in an arbitrary order.

NOTES

23 43 All components of an array have the same subtype. In particular, for an array of components that are one-dimensional arrays, this means that all components have the same bounds and hence the same length.

24 44 Each elaboration of an array_type_definition creates a distinct array type. A consequence of this is that each object whose object_declaration contains an array_type_definition is of its own unique type.

Examples

25 *Examples of type declarations with unconstrained array definitions:*

26
```
type Vector     is array(Integer  range <>) of Real;
type Matrix     is array(Integer  range <>, Integer range <>) of Real;
type Bit_Vector is array(Integer  range <>) of Boolean;
type Roman      is array(Positive range <>) of Roman_Digit; -- see 3.5.2
```

27 *Examples of type declarations with constrained array definitions:*

28
```
type Table    is array(1 .. 10) of Integer;
type Schedule is array(Day) of Boolean;
type Line     is array(1 .. Max_Line_Size) of Character;
```

29 *Examples of object declarations with array type definitions:*

30/2
```
{AI95-00433-01} Grid      : array(1 .. 80, 1 .. 100) of Boolean;
Mix       : array(Color range Red .. Green) of Boolean;
Msg_Table : constant array(Error_Code) of access constant String :=
      (Too_Big => new String'("Result too big"), Too_Small => ...);
Page      : array(Positive range <>) of Line :=  -- an array of arrays
  (1 | 50  => Line'(1 | Line'Last => '+', others => '-'),  -- see 4.3.3
   2 .. 49 => Line'(1 | Line'Last => '|', others => ' '));
    -- Page is constrained by its initial value to (1..50)
```

Extensions to Ada 83

30.a {*extensions to Ada 83*} The syntax rule for component_definition is modified to allow the reserved word **aliased**.

30.b The syntax rules for unconstrained_array_definition and constrained_array_definition are modified to use component_definition (instead of *component_*subtype_indication). The effect of this change is to allow the reserved word **aliased** before the component subtype_indication.

30.c A range in a discrete_subtype_definition may use arbitrary universal expressions for each bound (e.g. –1 .. 3+5), rather than strictly "implicitly convertible" operands. The subtype defined will still be a subtype of Integer.

Wording Changes from Ada 83

We introduce a new syntactic category, discrete_subtype_definition, as distinct from discrete_range. These two constructs have the same syntax, but their semantics are quite different (one defines a subtype, with a preference for Integer subtypes, while the other just selects a subrange of an existing subtype). We use this new syntactic category in **for** loops and entry families. 30.d

The syntax for index_constraint and discrete_range have been moved to their own subclause, since they are no longer used here. 30.e

The syntax rule for component_definition (formerly component_subtype_definition) is moved here from RM83-3.7. 30.f

Extensions to Ada 95

{*AI95-00230-01*} {*AI95-00406-01*} {*extensions to Ada 95*} Array components can have an anonymous access type. 30.g/2

{*AI95-00363-01*} The prohibition against unconstrained discriminated aliased components has been lifted. It has been replaced by a prohibition against the actual troublemakers: general access discriminant constraints (see 3.7.1). 30.h/2

Wording Changes from Ada 95

{*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Added wording to allow the elaboration of per-object constraints for constrained arrays. 30.i/2

3.6.1 Index Constraints and Discrete Ranges

An index_constraint determines the range of possible values for every index of an array subtype, and thereby the corresponding array bounds. 1

Syntax

index_constraint ::= (discrete_range {, discrete_range}) 2

discrete_range ::= *discrete*_subtype_indication | range 3

Name Resolution Rules

{*type of a discrete_range*} The type of a discrete_range is the type of the subtype defined by the subtype_indication, or the type of the range. {*expected type (index_constraint discrete_range)* [partial]} For an index_constraint, each discrete_range shall resolve to be of the type of the corresponding index. 4

Discussion: In Ada 95, index_constraints only appear in a subtype_indication; they no longer appear in constrained_array_definitions. 4.a

Legality Rules

An index_constraint shall appear only in a subtype_indication whose subtype_mark denotes either an unconstrained array subtype, or an unconstrained access subtype whose designated subtype is an unconstrained array subtype; in either case, the index_constraint shall provide a discrete_range for each index of the array type. 5

Static Semantics

{*bounds (of a discrete_range)*} A discrete_range defines a range whose bounds are given by the range, or by the range of the subtype defined by the subtype_indication. 6

Dynamic Semantics

{*compatibility (index constraint with a subtype)* [partial]} An index_constraint is *compatible* with an unconstrained array subtype if and only if the index range defined by each discrete_range is compatible (see 3.5) with the corresponding index subtype. {*null array*} If any of the discrete_ranges defines a null range, any array thus constrained is a *null array*, having no components. {*satisfies (an index constraint)* 7 [partial]} An array value *satisfies* an index_constraint if at each index position the array value and the index_constraint have the same index bounds.

7.a **Ramification:** There is no need to define compatibility with a constrained array subtype, because one is not allowed to constrain it again.

8 {*elaboration (index_constraint)* [partial]} The elaboration of an index_constraint consists of the evaluation of the discrete_range(s), in an arbitrary order. {*evaluation (discrete_range)* [partial]} The evaluation of a discrete_range consists of the elaboration of the subtype_indication or the evaluation of the range.

NOTES

9 45 The elaboration of a subtype_indication consisting of a subtype_mark followed by an index_constraint checks the compatibility of the index_constraint with the subtype_mark (see 3.2.2).

10 46 Even if an array value does not satisfy the index constraint of an array subtype, Constraint_Error is not raised on conversion to the array subtype, so long as the length of each dimension of the array value and the array subtype match. See 4.6.

Examples

11 *Examples of array declarations including an index constraint:*

12
```
Board     : Matrix(1 .. 8,  1 .. 8);  -- see 3.6
Rectangle : Matrix(1 .. 20, 1 .. 30);
Inverse   : Matrix(1 .. N,  1 .. N);  -- N need not be static
```

13
```
Filter    : Bit_Vector(0 .. 31);
```

14 *Example of array declaration with a constrained array subtype:*

15
```
My_Schedule : Schedule;  -- all arrays of type Schedule have the same bounds
```

16 *Example of record type with a component that is an array:*

17
```
type Var_Line(Length : Natural) is
   record
      Image : String(1 .. Length);
   end record;
```

18
```
Null_Line : Var_Line(0);  -- Null_Line.Image is a null array
```

Extensions to Ada 83

18.a {*extensions to Ada 83*} We allow the declaration of a variable with a nominally unconstrained array subtype, so long as it has an initialization expression to determine its bounds.

Wording Changes from Ada 83

18.b We have moved the syntax for index_constraint and discrete_range here since they are no longer used in constrained_array_definitions. We therefore also no longer have to describe the (special) semantics of index_constraints and discrete_ranges that appear in constrained_array_definitions.

18.c The rules given in RM83-3.6.1(5,7-10), which define the bounds of an array object, are redundant with rules given elsewhere, and so are not repeated here. RM83-3.6.1(6), which requires that the (nominal) subtype of an array variable be constrained, no longer applies, so long as the variable is explicitly initialized.

3.6.2 Operations of Array Types

Legality Rules

1 [The argument N used in the attribute_designators for the N-th dimension of an array shall be a static expression of some integer type.] The value of N shall be positive (nonzero) and no greater than the dimensionality of the array.

Static Semantics

{*8652/0006*} {*AI95-00030-01*} The following attributes are defined for a prefix A that is of an array type [(after any implicit dereference)], or denotes a constrained array subtype: 2/1

Ramification: These attributes are not defined if A is a subtype-mark for an access-to-array subtype. They are defined (by implicit dereference) for access-to-array values. 2.a

A'First — A'First denotes the lower bound of the first index range; its type is the corresponding index type. 3

A'First(N) — A'First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type. 4

A'Last — A'Last denotes the upper bound of the first index range; its type is the corresponding index type. 5

A'Last(N) — A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type. 6

A'Range — A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once. 7

A'Range(N) — A'Range(N) is equivalent to the range A'First(N) .. A'Last(N), except that the prefix A is only evaluated once. 8

A'Length — A'Length denotes the number of values of the first index range (zero for a null range); its type is *universal_integer*. 9

A'Length(N) — A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is *universal_integer*. 10

Implementation Advice

An implementation should normally represent multidimensional arrays in row-major order, consistent with the notation used for multidimensional array aggregates (see 4.3.3). However, if a **pragma** Convention(Fortran, ...) applies to a multidimensional array type, then column-major order should be used instead (see B.5, “Interfacing with Fortran”). 11

Implementation Advice: Multidimensional arrays should be represented in row-major order, unless the array has convention Fortran. 11.a/2

NOTES

47 The attribute_references A'First and A'First(1) denote the same value. A similar relation exists for the attribute_references A'Last, A'Range, and A'Length. The following relation is satisfied (except for a null array) by the above attributes if the index type is an integer type: 12

```
A'Length(N) = A'Last(N) - A'First(N) + 1
```
13

48 An array type is limited if its component type is limited (see 7.5). 14

49 {*predefined operations (of an array type)* [partial]} The predefined operations of an array type include the membership tests, qualification, and explicit conversion. If the array type is not limited, they also include assignment and the predefined equality operators. For a one-dimensional array type, they include the predefined concatenation operators (if nonlimited) and, if the component type is discrete, the predefined relational operators; if the component type is boolean, the predefined logical operators are also included. 15

50 {*AI95-00287-01*} A component of an array can be named with an indexed_component. A value of an array type can be specified with an array_aggregate. For a one-dimensional array type, a slice of the array can be named; also, string literals are defined if the component type is a character type. 16/2

Examples

17 *Examples (using arrays declared in the examples of subclause 3.6.1):*

18
```
--  Filter'First      =   0   Filter'Last       =  31   Filter'Length =  32
--  Rectangle'Last(1) =  20   Rectangle'Last(2) =  30
```

3.6.3 String Types

Static Semantics

1 {*string type*} A one-dimensional array type whose component type is a character type is called a *string* type.

2/2 {*AI95-00285-01*} [There are three predefined string types, String, Wide_String, and Wide_Wide_String, each indexed by values of the predefined subtype Positive; these are declared in the visible part of package Standard:]

3
```
[subtype Positive is Integer range 1 .. Integer'Last;
```

4/2
```
{AI95-00285-01} type String is array(Positive range <>) of Character;
type Wide_String is array(Positive range <>) of Wide_Character;
type Wide_Wide_String is array(Positive range <>) of Wide_Wide_Character;
]
```

NOTES

5 51 String literals (see 2.6 and 4.2) are defined for all string types. The concatenation operator & is predefined for string types, as for all nonlimited one-dimensional array types. The ordering operators <, <=, >, and >= are predefined for string types, as for all one-dimensional discrete array types; these ordering operators correspond to lexicographic order (see 4.5.2).

Examples

6 *Examples of string objects:*

7
```
Stars      : String(1 .. 120) := (1 .. 120 => '*' );
Question   : constant String  := "How many characters?";
                                        -- Question'First = 1, Question'Last = 20
                                        -- Question'Length = 20 (the number of characters)
```

8
```
Ask_Twice  : String  := Question & Question;       -- constrained to (1..40)
Ninety_Six : constant Roman   := "XCVI";  -- see 3.5.2 and 3.6
```

Inconsistencies With Ada 83

8.a {*inconsistencies with Ada 83*} The declaration of Wide_String in Standard hides a use-visible declaration with the same defining_identifier. In rare cases, this might result in an inconsistency between Ada 83 and Ada 95.

Incompatibilities With Ada 83

8.b {*incompatibilities with Ada 83*} Because both String and Wide_String are always directly visible, an expression like

8.c
```
"a" < "bc"
```

8.d is now ambiguous, whereas in Ada 83 both string literals could be resolved to type String.

Extensions to Ada 83

8.e {*extensions to Ada 83*} The type Wide_String is new (though it was approved by ARG for Ada 83 compilers as well).

Wording Changes from Ada 83

8.f We define the term *string type* as a natural analogy to the term *character type*.

Inconsistencies With Ada 95

8.g/2 {*AI95-00285-01*} {*inconsistencies with Ada 95*} The declaration of Wide_Wide_String in Standard hides a use-visible declaration with the same defining_identifier. In the (very) unlikely event that an Ada 95 program had depended on such a use-visible declaration, and the program remains legal after the substitution of Standard.Wide_Wide_String, the meaning of the program will be different.

Extensions to Ada 95

{*AI95-00285-01*} {*extensions to Ada 95*} The type Wide_Wide_String is new. 8.h/2

3.7 Discriminants

{*AI95-00251-01*} {*AI95-00326-01*} [{*discriminant*} {*type parameter: See discriminant*} {*parameter: See also discriminant*} A composite type (other than an array or interface type) can have discriminants, which parameterize the type. A known_discriminant_part specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An unknown_discriminant_part in the declaration of a view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a view are indefinite subtypes.] 1/2

Glossary entry: {*Discriminant*} A discriminant is a parameter for a composite type. It can control, for example, the bounds of a component of the type if the component is an array. A discriminant for a task type can be used to pass data to a task of the type upon creation. 1.a/2

Discussion: {*AI95-00114-01*} {*unknown discriminants* [partial]} {*discriminants (unknown)* [partial]} A view of a type, and all subtypes of the view, have *unknown discriminants* when the number or names of the discriminants, if any, are unknown at the point of the type declaration for the view. A discriminant_part of (<>) is used to indicate unknown discriminants. 1.b/2

Language Design Principles

{*AI95-00402-01*} When an access discriminant is initialized at the time of object creation with an allocator of an anonymous type, the allocated object and the object with the discriminant are tied together for their lifetime. They should be allocated out of the same storage pool, and then at the end of the lifetime of the enclosing object, finalized and reclaimed together. In this case, the allocated object is called a coextension (see 3.10.2). 1.c/2

Discussion: The above principle when applied to a nonlimited type implies that such an object may be copied only to a shorter-lived object, because attempting to assign it to a longer-lived object would fail because the access discriminants would not match. In a copy, the lifetime connection between the enclosing object and the allocated object does not exist. The allocated object is tied in the above sense only to the original object. Other copies have only secondary references to it. 1.d/2

Note that when an allocator appears as a constraint on an access discriminant in a subtype_indication that is elaborated independently from object creation, no such connection exists. For example, if a named constrained subtype is declared via "**subtype** Constr **is** Rec(Acc_Discrim => **new** T);" or if such an allocator appears in the subtype_indication for a component, the allocator is evaluated when the subtype_indication is elaborated, and hence its lifetime is typically longer than the objects or components that will later be subject to the constraint. In these cases, the allocated object should not be reclaimed until the subtype_indication goes out of scope. 1.e/2

Syntax

discriminant_part ::= unknown_discriminant_part | known_discriminant_part 2

unknown_discriminant_part ::= (<>) 3

known_discriminant_part ::= 4
  (discriminant_specification {; discriminant_specification})

{*AI95-00231-01*} discriminant_specification ::= 5/2
  defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]

default_expression ::= expression 6

Name Resolution Rules

{*expected type (discriminant default_expression)* [partial]} The expected type for the default_expression of a discriminant_specification is that of the corresponding discriminant. 7

Legality Rules

8/2 {*8652/0007*} {*AI95-00098-01*} {*AI95-00251-01*} A discriminant_part is only permitted in a declaration for a composite type that is not an array or interface type [(this includes generic formal types)]. A type declared with a known_discriminant_part is called a *discriminated* type,{*discriminated type*} as is a type that inherits (known) discriminants.

8.a **Implementation Note:** Discriminants on array types were considered, but were omitted to ease (existing) implementations.

8.b **Discussion:** Note that the above definition for "discriminated type" does not include types declared with an unknown_discriminant_part. This seems consistent with Ada 83, where such types (in a generic formal part) would not be considered discriminated types. Furthermore, the full type for a type with unknown discriminants need not even be composite, much less have any discriminants.

8.b.1/1 {*8652/0007*} {*AI95-00098-01*} On the other hand, unknown_discriminant_parts cannot be applied to type declarations that cannot have a known_discriminant_part. There is no point in having unknown discriminants on a type that can never have discriminants (for instance, a formal modular type), even when these are allowed syntactically.

9/2 {*AI95-00231-01*} {*AI95-00254-01*} The subtype of a discriminant may be defined by an optional null_exclusion and a subtype_mark, in which case the subtype_mark shall denote a discrete or access subtype, or it may be defined by an access_definition. {*access discriminant*} A discriminant that is defined by an access_definition is called an *access discriminant* and is of an anonymous access type.

9.a/2 *This paragraph was deleted.*{*AI95-00230-01*}

9.b **Reason:** Note that discriminants of a named access type are not considered "access discriminants." Similarly, "access parameter" only refers to a formal parameter defined by an access_definition.

9.1/2 {*AI95-00402-01*} Default_expressions shall be provided either for all or for none of the discriminants of a known_discriminant_part. No default_expressions are permitted in a known_discriminant_part in a declaration of a tagged type [or a generic formal type].

9.c/2 **Reason:** The all-or-none rule is related to the rule that a discriminant constraint shall specify values for all discriminants. One could imagine a different rule that allowed a constraint to specify only some of the discriminants, with the others provided by default. Having defaults for discriminants has a special significance — it allows objects of the type to be unconstrained, with the discriminants alterable as part of assigning to the object.

9.d/2 Defaults for discriminants of tagged types are disallowed so that every object of a tagged type is constrained, either by an explicit constraint, or by its initial discriminant values. This substantially simplifies the semantic rules and the implementation of inherited dispatching operations. For generic formal types, the restriction simplifies the type matching rules. If one simply wants a "default" value for the discriminants, a constrained subtype can be declared for future use.

10/2 {*AI95-00230-01*} {*AI95-00402-01*} {*AI95-00419-01*} A discriminant_specification for an access discriminant may have a default_expression only in the declaration for a task or protected type, or for a type that is a descendant of an explicitly limited record type. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.{*generic contract issue* [partial]}

10.a/2 **Discussion:** This rule implies that a type can have a default for an access discriminant if the type is limited, but not if the only reason it's limited is because of a limited component. Compare with the definition of limited type in 7.5. Also, recall that a "descendant' includes the type itself, so an explicitly limited record type can have defaults.

10.b/2 *This paragraph was deleted.*

10.c/2 **Reason:** {*AI95-00230-01*} We considered the following rules for access discriminants:

10.d
- If a type has an access discriminant, this automatically makes it limited, just like having a limited component automatically makes a type limited. This was rejected because it decreases program readability, and because it seemed error prone (two bugs in a previous version of the RM9X were attributable to this rule).

10.e/2
- A type with an access discriminant shall be limited. This is equivalent to the rule we actually chose for Ada 95, except that it allows a type to have an access discriminant if it is limited just because of a limited component. For example, any record containing a task would be allowed to have an access discriminant, whereas the actual rule requires "**limited record**". This rule was also rejected due to readability concerns, and because would interact badly with the rules for limited types that "become nonlimited".

- A type may have an access discriminant if it is a limited partial view, or a task, protected, or explicitly limited record type. This was the rule chosen for Ada 95. 10.f/2
- Any type may have an access discriminant. For nonlimited type, there is no special accessibility for access discriminants; they're the same as any other anonymous access component. For a limited type, they have the special accessibility of Ada 95. However, this doesn't work because a limited partial view can have a nonlimited full view -- giving the two views different accessibility. 10.g/2
- Any type may have an access discriminant, as above. However, special accessibility rules only apply to types that are “really” limited (task, protected, and explicitly limited records). However, this breaks privacy; worse, Legality Rules depend on the definition of accessibility. 10.h/2
- Any type may have an access discriminant, as above. Limited types have special accessibility, while nonlimited types have normal accessibility. However, a limited partial view with an access discriminant can only be completed by a task, protected, or explicitly limited record type. That prevents accessibility from changing. A runtime accessibility check is required on generic formal types with access discriminants. However, changing between limited and nonlimited types would have far-reaching consequences for access discriminants — which is uncomfortable. 10.i/2
- Any type may have an access discriminant. All types have special accessibility. This was considered early during the Ada 9X process, but was dropped for “unpleasant complexities”, which unfortunately aren't recorded. It does seem that an accessibility check would be needed on assignment of such a type, to avoid copying an object with a discriminant pointing to a local object into a more global object (and thus creating a dangling pointer). 10.j/2
- Any type may have an access discriminant, but access discriminants cannot have defaults. All types have special accessibility. This gets rid of the problems on assignment (you couldn't change such a discriminant), but it would be horribly incompatible with Ada 95. 10.k/2
- Any type may have an access discriminant, but access discriminants may have defaults only if they are a “really” limited type. This is the rule chosen for Ada 2005, as it is not incompatible, and it doesn't require weird accessibility checks. 10.l/2

This paragraph was deleted.{*AI95-00402-01*} 11/2

For a type defined by a derived_type_definition, if a known_discriminant_part is provided in its declaration, then: 12

- The parent subtype shall be constrained; 13
- If the parent type is not a tagged type, then each discriminant of the derived type shall be used in the constraint defining the parent subtype; 14

  Implementation Note: This ensures that the new discriminant can share storage with an existing discriminant. 14.a

- If a discriminant is used in the constraint defining the parent subtype, the subtype of the discriminant shall be statically compatible (see 4.9.1) with the subtype of the corresponding parent discriminant. 15

  Reason: This ensures that on conversion (or extension via an extension aggregate) to a distantly related type, if the discriminants satisfy the target type's requirements they satisfy all the intermediate types' requirements as well. 15.a

  Ramification: There is no requirement that the new discriminant have the same (or any) default_expression as the parent's discriminant. 15.b

The type of the default_expression, if any, for an access discriminant shall be convertible to the anonymous access type of the discriminant (see 4.6). {*convertible (required)* [partial]} 16

Ramification: This requires convertibility of the designated subtypes. 16.a

Static Semantics

A discriminant_specification declares a discriminant; the subtype_mark denotes its subtype unless it is an access discriminant, in which case the discriminant's subtype is the anonymous access-to-variable subtype defined by the access_definition. 17

[For a type defined by a derived_type_definition, each discriminant of the parent type is either inherited, constrained to equal some new discriminant of the derived type, or constrained to the value of an expression.] {*corresponding discriminants*} When inherited or constrained to equal some new discriminant, the parent discriminant and the discriminant of the derived type are said to *correspond*. Two discriminants 18 also correspond if there is some common discriminant to which they both correspond. A discriminant corresponds to itself as well. {*specified discriminant*} If a discriminant of a parent type is constrained to a specific value by a derived_type_definition, then that discriminant is said to be *specified* by that derived_type_definition.

18.a **Ramification:** The correspondence relationship is transitive, symmetric, and reflexive. That is, if A corresponds to B, and B corresponds to C, then A, B, and C each corresponds to A, B, and C in all combinations.

19 {*depend on a discriminant (for a constraint or component_definition)*} A constraint that appears within the definition of a discriminated type *depends on a discriminant* of the type if it names the discriminant as a bound or discriminant value. A component_definition depends on a discriminant if its constraint depends on the discriminant, or on a discriminant that corresponds to it.

19.a **Ramification:** A constraint in a task_body is not considered to *depend* on a discriminant of the task type, even if it names it. It is only the constraints in the type definition itself that are considered dependents. Similarly for protected types.

20 {*depend on a discriminant (for a component)*} A component *depends on a discriminant* if:

21 • Its component_definition depends on the discriminant; or

21.a **Ramification:** A component does *not* depend on a discriminant just because its default_expression refers to the discriminant.

22 • It is declared in a variant_part that is governed by the discriminant; or

23 • It is a component inherited as part of a derived_type_definition, and the constraint of the *parent*_subtype_indication depends on the discriminant; or

23.a **Reason:** When the parent subtype depends on a discriminant, the parent part of the derived type is treated like a discriminant-dependent component.

23.b **Ramification:** Because of this rule, we don't really need to worry about “corresponding” discriminants, since all the inherited components will be discriminant-dependent if there is a new known_discriminant_part whose discriminants are used to constrain the old discriminants.

24 • It is a subcomponent of a component that depends on the discriminant.

24.a **Reason:** The concept of discriminant-dependent (sub)components is primarily used in various rules that disallow renaming or 'Access, or specify that certain discriminant-changing assignments are erroneous. The goal is to allow implementations to move around or change the size of discriminant-dependent subcomponents upon a discriminant-changing assignment to an enclosing object. The above definition specifies that all subcomponents of a discriminant-dependent component or parent part are themselves discriminant-dependent, even though their presence or size does not in fact depend on a discriminant. This is because it is likely that they will move in a discriminant-changing assignment if they are a component of one of several discriminant-dependent parts of the same record.

25 Each value of a discriminated type includes a value for each component of the type that does not depend on a discriminant[; this includes the discriminants themselves]. The values of discriminants determine which other component values are present in the value of the discriminated type.

25.a **To be honest:** Which values are present might depend on discriminants of some ancestor type that are constrained in an intervening derived_type_definition. That's why we say "values of discriminants" instead of "values of *the* discriminants" — a subtle point.

26 {*known discriminants*} {*discriminants (known)*} {*constrained (subtype)*} {*unconstrained (subtype)*} A type declared with a known_discriminant_part is said to have *known discriminants*; its first subtype is unconstrained. {*unknown discriminants*} {*discriminants (unknown)*} A type declared with an unknown_discriminant_part is said to have *unknown discriminants*. A type declared without a discriminant_part has no discriminants, unless it is a derived type; if derived, such a type has the same sort of discriminants (known, unknown, or none) as its parent (or ancestor) type. A tagged class-wide type also has unknown discriminants. {*class-wide type*} {*indefinite subtype*} [Any subtype of a type with unknown discriminants is an unconstrained and indefinite subtype (see 3.2 and 3.3).]

26.a/2 **Discussion:** {*AI95-00114-01*} An unknown_discriminant_part “(<>)” is only permitted in the declaration of a (generic or nongeneric) private type, private extension, incomplete type, or formal derived type. Hence, only such types, descendants thereof, and class-wide types can have unknown discriminants. An unknown_discriminant_part is used to indicate that the corresponding actual or full type might have discriminants without defaults, or be an unconstrained array subtype. Tagged class-wide types are also considered to have unknown discriminants because discriminants can be added by type extensions, so the total number of discriminants of any given value of a tagged class-wide type is not known at compile time.

26.b/2 {*AI95-00287-01*} A subtype with unknown discriminants is indefinite, and hence an object of such a subtype needs explicit initialization. A limited private type with unknown discriminants is “extremely” limited; objects of such a type can be initialized only by subprograms (either procedures with a parameter of the type, or a function returning the type) declared in the package. Subprograms declared elsewhere can operate on and even return the type, but they can only initialize the object by calling (ultimately) a subprogram in the package declaring the type. Such a type is useful for keeping complete control over object creation within the package declaring the type.

26.c A partial view of a type might have unknown discriminants, while the full view of the same type might have known, unknown, or no discriminants.

Dynamic Semantics

27/2 {*AI95-00230-01*} {*AI95-00416-01*} For an access discriminant, its access_definition is elaborated when the value of the access discriminant is defined: by evaluation of its default_expression, by elaboration of a discriminant_constraint, or by an assignment that initializes the enclosing object. {*implicit subtype conversion (access discriminant)* [partial]}

27.a/2 **Ramification:** {*AI95-00231-01*} {*AI95-00416-01*} The conversion of the expression defining the access discriminant to the anonymous access type raises Program_Error for an object created by an allocator of an access type T, if the initial value is an access parameter that designates a view whose accessibility level is deeper than that of T.

NOTES

28 52 If a discriminated type has default_expressions for its discriminants, then unconstrained variables of the type are permitted, and the values of the discriminants can be changed by an assignment to such a variable. If defaults are not provided for the discriminants, then all variables of the type are constrained, either by explicit constraint or by their initial value; the values of the discriminants of such a variable cannot be changed after initialization.

28.a **Discussion:** This connection between discriminant defaults and unconstrained variables can be a source of confusion. For Ada 95, we considered various ways to break the connection between defaults and unconstrainedness, but ultimately gave up for lack of a sufficiently simple and intuitive alternative.

28.b {*mutable*} An unconstrained discriminated subtype with defaults is called a *mutable* subtype, and a variable of such a subtype is called a mutable variable, because the discriminants of such a variable can change. There are no mutable arrays (that is, the bounds of an array object can never change), because there is no way in the language to define default values for the bounds. Similarly, there are no mutable class-wide subtypes, because there is no way to define the default tag, and defaults for discriminants are not allowed in the tagged case. Mutable tags would also require a way for the maximum possible size of such a class-wide subtype to be known. (In some implementations, all mutable variables are allocated with the maximum possible size. This approach is appropriate for real-time applications where implicit use of the heap is inappropriate.)

29 53 The default_expression for a discriminant of a type is evaluated when an object of an unconstrained subtype of the type is created.

30 54 Assignment to a discriminant of an object (after its initialization) is not allowed, since the name of a discriminant is a constant; neither assignment_statements nor assignments inherent in passing as an **in out** or **out** parameter are allowed. Note however that the value of a discriminant can be changed by assigning to the enclosing object, presuming it is an unconstrained variable.

30.a/2 **Discussion:** {*AI95-00114-01*} An unknown_discriminant_part is permitted only in the declaration of a private type (including generic formal private), private extension, incomplete type, or generic formal derived type. These are the things that will have a corresponding completion or generic actual, which will either define the discriminants, or say there are none. The (<>) indicates that the actual/full subtype might be an indefinite subtype. An unknown_discriminant_part is not permitted in a normal untagged derived type declaration, because there is no separate full type declaration for such a type. Note that (<>) allows unconstrained array bounds; those are somewhat like undefaulted discriminants.

30.b For a derived type, either the discriminants are inherited as is, or completely respecified in a new discriminant_part. In this latter case, each discriminant of the parent type shall be constrained, either to a specific value, or to equal one of the new discriminants. Constraining a parent type's discriminant to equal one of the new discriminants is like a renaming of the discriminant, except that the subtype of the new discriminant can be more restrictive than that of the parent's one. In any case, the new discriminant can share storage with the parent's discriminant.

31 55 A discriminant that is of a named access type is not called an access discriminant; that term is used only for discriminants defined by an access_definition.

Examples

32 *Examples of discriminated types:*

33
```
type Buffer(Size : Buffer_Size := 100)  is         -- see 3.5.4
   record
      Pos   : Buffer_Size := 0;
      Value : String(1 .. Size);
   end record;
```

34
```
type Matrix_Rec(Rows, Columns : Integer) is
   record
      Mat : Matrix(1 .. Rows, 1 .. Columns);       -- see 3.6
   end record;
```

35
```
type Square(Side : Integer) is new
   Matrix_Rec(Rows => Side, Columns => Side);
```

36
```
type Double_Square(Number : Integer) is
   record
      Left  : Square(Number);
      Right : Square(Number);
   end record;
```

37/2
```
{AI95-00433-01} task type Worker(Prio : System.Priority; Buf : access Buffer) is
   -- discriminants used to parameterize the task type (see 9.1)
   pragma Priority(Prio);  -- see D.1
   entry Fill;
   entry Drain;
end Worker;
```

Extensions to Ada 83

37.a {*extensions to Ada 83*} The syntax for a discriminant_specification is modified to allow an *access discriminant*, with a type specified by an access_definition (see 3.10).

37.b/2 {*AI95-00251-01*} Discriminants are allowed on all composite types other than array and interface types.

37.c Discriminants may be of an access type.

Wording Changes from Ada 83

37.d Discriminant_parts are not elaborated, though an access_definition is elaborated when the discriminant is initialized.

Extensions to Ada 95

37.e/2 {*AI95-00230-01*} {*AI95-00402-01*} {*AI95-00416-01*} Access discriminants (anonymous access types used as a discriminant) can be used on any type allowing discriminants. Defaults aren't allowed on discriminants of non-limited types, however, so that accessibility problems don't happen on assignment.

37.f/2 {*AI95-00231-01*} null_exclusion can be used in the declaration of a discriminant.

Wording Changes from Ada 95

37.g/2 {*8652/0007*} {*AI95-00098-01*} **Corrigendum:** The wording was clarified so that types that cannot have discriminants cannot have an unknown_discriminant_part.

37.h/2 {*AI95-00251-01*} Added wording to prevent interfaces from having discriminants. We don't want interfaces to have any components.

37.i/2 {*AI95-00254-01*} Removed wording which implied or required an access discriminant to have an access-to-object type (anonymous access types can now be access-to-subprogram types as well).

37.j/2 {*AI95-00326-01*} Fixed the wording of the introduction to this clause to reflect that both incomplete and partial views can have unknown discriminants. That was always true, but for some reason this wording specified partial views.

37.k/2 {*AI95-00419-01*} Changed the wording to use the new term “explicitly limited record”, which makes the intent much clearer (and eliminates confusion with derived types that happen to contain the reserved word **limited**.

3.7.1 Discriminant Constraints

1 A discriminant_constraint specifies the values of the discriminants for a given discriminated type.

Language Design Principles

1.a The rules in this clause are intentionally parallel to those given in Record Aggregates.

Syntax

2 discriminant_constraint ::=
  (discriminant_association {, discriminant_association})

3 discriminant_association ::=
  [*discriminant*_selector_name {| *discriminant*_selector_name} =>] expression

4 {*named discriminant association*} A discriminant_association is said to be *named* if it has one or more *discriminant*_selector_names; {*positional discriminant association*} it is otherwise said to be *positional*. In a discriminant_constraint, any positional associations shall precede any named associations.

Name Resolution Rules

5 Each selector_name of a named discriminant_association shall resolve to denote a discriminant of the subtype being constrained; {*associated discriminants (of a named discriminant_association)*} the discriminants so named are the *associated discriminants* of the named association. {*associated discriminants (of a positional discriminant_association)*} For a positional association, the *associated discriminant* is the one whose discriminant_specification occurred in the corresponding position in the known_discriminant_-part that defined the discriminants of the subtype being constrained.

6 {*expected type (discriminant_association expression)* [partial]} The expected type for the expression in a discriminant_association is that of the associated discriminant(s).

Legality Rules

7/2 {*8652/0008*} {*AI95-00168-01*} {*AI95-00363-01*} A discriminant_constraint is only allowed in a subtype_indication whose subtype_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of an access subtype, a discriminant_constraint is illegal if the designated type has a partial view that is constrained or, for a general access subtype, has default_expressions for its discriminants. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. In a generic body, this rule is checked presuming all formal access types of the generic might be general access types, and all untagged discriminated formal types of the generic might have default_expressions for their discriminants. {*generic contract issue* [partial]}

7.a.1/2 *This paragraph was deleted.*{*8652/0008*} {*AI95-00168-01*} {*AI95-00363-01*}

7.a/2 **Reason:** {*AI95-00363-01*} The second rule is necessary to prevent objects from changing so that they no longer match their constraint. In Ada 95, we attempted to prevent this by banning every case where an aliased object could be unconstrained or be changed by an enclosing assignment. New ways to cause this problem were being discovered frequently, meaning that new rules had to be dreamed up to cover them. Meanwhile, aliased objects and components were getting more and more limited. In Ada 2005, we sweep away all of that cruft and replace it by a simple rule “thou shalt not create an access subtype that can point to an item whose discriminants can be changed by assignment”.

8 A named discriminant_association with more than one selector_name is allowed only if the named discriminants are all of the same type. A discriminant_constraint shall provide exactly one value for each discriminant of the subtype being constrained.

9 The expression associated with an access discriminant shall be of a type convertible to the anonymous access type. {*convertible (required)* [partial]}

9.a **Ramification:** This implies both convertibility of designated types, and static accessibility. This implies that if an object of type T with an access discriminant is created by an allocator for an access type A, then it requires that the type of the expression associated with the access discriminant have an accessibility level that is not statically deeper than that of A. This is to avoid dangling references.

Dynamic Semantics

10 {*compatibility (discriminant constraint with a subtype)* [partial]} A discriminant_constraint is *compatible* with an unconstrained discriminated subtype if each discriminant value belongs to the subtype of the corresponding discriminant.

10.a **Ramification:** The "dependent compatibility check" has been eliminated in Ada 95. Any checking on subcomponents is performed when (and if) an object is created.

10.b **Discussion:** There is no need to define compatibility with a constrained discriminated subtype, because one is not allowed to constrain it again.

11 {*satisfies (a discriminant constraint)* [partial]} A composite value *satisfies* a discriminant constraint if and only if each discriminant of the composite value has the value imposed by the discriminant constraint.

12 {*elaboration (discriminant_constraint)* [partial]} For the elaboration of a discriminant_constraint, the expressions in the discriminant_associations are evaluated in an arbitrary order and converted to the type of the associated discriminant (which might raise Constraint_Error — see 4.6); the expression of a named association is evaluated (and converted) once for each associated discriminant. {*implicit subtype conversion (discriminant values)* [partial]} The result of each evaluation and conversion is the value imposed by the constraint for the associated discriminant.

12.a **Reason:** We convert to the type, not the subtype, so that the definition of compatibility of discriminant constraints is not vacuous.

NOTES

13 56 The rules of the language ensure that a discriminant of an object always has a value, either from explicit or implicit initialization.

13.a **Discussion:** Although it is illegal to constrain a class-wide tagged subtype, it is possible to have a partially constrained class-wide subtype: If the subtype S is defined by T(A => B), then S'Class is partially constrained in the sense that objects of subtype S'Class have to have discriminants corresponding to A equal to B, but there can be other discriminants defined in extensions that are not constrained to any particular value.

Examples

14 *Examples (using types declared above in clause 3.7):*

15
```
Large   : Buffer(200);  -- constrained, always 200 characters
                        --  (explicit discriminant value)
Message : Buffer;       -- unconstrained, initially 100 characters
                        --  (default discriminant value)
Basis   : Square(5);    -- constrained, always 5 by 5
Illegal : Square;       -- illegal, a Square has to be constrained
```

Inconsistencies With Ada 83

15.a {*inconsistencies with Ada 83*} Dependent compatibility checks are no longer performed on subtype declaration. Instead they are deferred until object creation (see 3.3.1). This is upward compatible for a program that does not raise Constraint_Error.

Wording Changes from Ada 83

15.b Everything in RM83-3.7.2(7-12), which specifies the initial values for discriminants, is now redundant with 3.3.1, 6.4.1, 8.5.1, and 12.4. Therefore, we don't repeat it here. Since the material is largely intuitive, but nevertheless complicated to state formally, it doesn't seem worth putting it in a "NOTE."

Incompatibilities With Ada 95

15.c/2 {*8652/0008*} {*AI95-00168-01*} {*AI95-00363-01*} {*incompatibilities with Ada 95*} The Corrigendum added a restriction on discriminant_constraints for general access subtypes. Such constraints are prohibited if the designated type can be treated as constrained somewhere in the program. Ada 2005 goes further and prohibits such discriminant_constraints if the designated type has (or might have, in the case of a formal type) defaults for its discriminants. The use of general access subtypes is rare, and this eliminates a boatload of problems that required many restrictions on the use of aliased objects and components (now lifted). Similarly, Ada 2005 prohibits discriminant_constraints on any access type whose designated type has a partial view that is constrained. Such a type will not be constrained in the heap to avoid privacy problems. Again, the use of such subtypes is rare (they can only happen within the package and its child units).

3.7.2 Operations of Discriminated Types

1 [If a discriminated type has default_expressions for its discriminants, then unconstrained variables of the type are permitted, and the discriminants of such a variable can be changed by assignment to the variable. For a formal parameter of such a type, an attribute is provided to determine whether the corresponding actual parameter is constrained or unconstrained.]

Static Semantics

2 For a prefix A that is of a discriminated type [(after any implicit dereference)], the following attribute is defined:

3 A'Constrained

Yields the value True if A denotes a constant, a value, or a constrained variable, and False otherwise.

3.a **Implementation Note:** This attribute is primarily used on parameters, to determine whether the discriminants can be changed as part of an assignment. The Constrained attribute is statically True for **in** parameters. For **in out** and **out** parameters of a discriminated type, the value of this attribute needs to be passed as an implicit parameter, in general. However, if the type does not have defaults for its discriminants, the attribute is statically True, so no implicit parameter is needed. Parameters of a limited type with defaulted discriminants need this implicit parameter, unless there are no nonlimited views, because they might be passed to a subprogram whose body has visibility on a nonlimited view of the type, and hence might be able to assign to the object and change its discriminants.

Erroneous Execution

4 {*erroneous execution (cause)* [partial]} The execution of a construct is erroneous if the construct has a constituent that is a name denoting a subcomponent that depends on discriminants, and the value of any of these discriminants is changed by this execution between evaluating the name and the last use (within this execution) of the subcomponent denoted by the name.

4.a **Ramification:** This rule applies to assignment_statements, calls (except when the discriminant-dependent subcomponent is an **in** parameter passed by copy), indexed_components, and slices. Ada 83 only covered the first two cases. AI83-00585 pointed out the situation with the last two cases. The cases of object_renaming_declarations and generic formal **in out** objects are handled differently, by disallowing the situation at compile time.

Extensions to Ada 83

4.b/1 {*extensions to Ada 83*} For consistency with other attributes, we are allowing the prefix of Constrained to be a value as well as an object of a discriminated type, and also an implicit dereference. These extensions are not important capabilities, but there seems no reason to make this attribute different from other similar attributes. We are curious what most Ada 83 compilers do with F(1).X'Constrained.

4.c We now handle in a general way the cases of erroneousness identified by AI83-00585, where the prefix of an indexed_component or slice is discriminant-dependent, and the evaluation of the index or discrete range changes the value of a discriminant.

Wording Changes from Ada 83

4.d We have moved all discussion of erroneous use of names that denote discriminant-dependent subcomponents to this subclause. In Ada 83, it used to appear separately under assignment_statements and subprogram calls.

3.8 Record Types

1 {*record*} {*record type*} A record object is a composite object consisting of named components. The value of a record object is a composite value consisting of the values of the components. {*structure: See record type*}

Syntax

2 record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition

3
```
record_definition ::=
    record
       component_list
    end record
  | null record
```

4
```
component_list ::=
      component_item {component_item}
   | {component_item} variant_part
   |  null;
```

5/1 {*8652/0009*} {*AI95-00137-01*} component_item ::= component_declaration | aspect_clause

6
```
component_declaration ::=
   defining_identifier_list : component_definition [:= default_expression];
```

Name Resolution Rules

7 {*expected type (component_declaration default_expression)* [partial]} The expected type for the default_expression, if any, in a component_declaration is the type of the component.

Legality Rules

8/2 *This paragraph was deleted.*{*AI95-00287-01*}

9/2 {*AI95-00366-01*} {*components (of a record type)* [partial]} Each component_declaration declares a component of the record type. Besides components declared by component_declarations, the components of a record type include any components declared by discriminant_specifications of the record type declaration. [The identifiers of all components of a record type shall be distinct.]

9.a **Proof:** The identifiers of all components of a record type have to be distinct because they are all declared immediately within the same declarative region. See Section 8.

10 Within a type_declaration, a name that denotes a component, protected subprogram, or entry of the type is allowed only in the following cases:

11 • A name that denotes any component, protected subprogram, or entry is allowed within a representation item that occurs within the declaration of the composite type.

12 • A name that denotes a noninherited discriminant is allowed within the declaration of the type, but not within the discriminant_part. If the discriminant is used to define the constraint of a component, the bounds of an entry family, or the constraint of the parent subtype in a derived_type_definition then its name shall appear alone as a direct_name (not as part of a larger expression or expanded name). A discriminant shall not be used to define the constraint of a scalar component.{*discriminant (use in a record definition)*}

12.a **Reason:** The penultimate restriction simplifies implementation, and allows the outer discriminant and the inner discriminant or bound to possibly share storage.

12.b **Ramification:** Other rules prevent such a discriminant from being an inherited one.

Reason: The last restriction is inherited from Ada 83. The restriction is not really necessary from a language design point of view, but we did not remove it, in order to avoid unnecessary changes to existing compilers. 12.c

Discussion: Note that a discriminant can be used to define the constraint for a component that is of an access-to-composite type. 12.d

Reason: {*AI95-00373-01*} The above rules, and a similar one in 6.1 for formal parameters, are intended to allow initializations of components or parameters to occur in a (nearly) arbitrary order — whatever order is most efficient (subject to the restrictions of 3.3.1), since one default_expression cannot depend on the value of another one. They also prevent circularities. 12.e/2

Ramification: Inherited discriminants are not allowed to be denoted, except within representation items. However, the *discriminant_*selector_name of the parent subtype_indication is allowed to denote a discriminant of the parent. 12.f

If the name of the current instance of a type (see 8.6) is used to define the constraint of a component, then it shall appear as a direct_name that is the prefix of an attribute_reference whose result is of an access type, and the attribute_reference shall appear alone. 13

Reason: This rule allows T'Access or T'Unchecked_Access, but disallows, for example, a range constraint (1..T'Size). Allowing things like (1..T'Size) would mean that a per-object constraint could affect the size of the object, which would be bad. 13.a

Static Semantics

{*AI95-00318-02*} {*explicitly limited record*} {*record (explicitly limited)*} If a record_type_definition includes the reserved word **limited**, the type is called an *explicitly limited record* type. 13.1/2

{*nominal subtype (of a record component)* [partial]} The component_definition of a component_declaration defines the (nominal) subtype of the component. If the reserved word **aliased** appears in the component_definition, then the component is aliased (see 3.10). 14

{*null record*} If the component_list of a record type is defined by the reserved word **null** and there are no discriminants, then the record type has no components and all records of the type are *null records*. A record_definition of **null record** is equivalent to **record null; end record**. 15

Ramification: This short-hand is available both for declaring a record type and a record extension — see 3.9.1. 15.a

Dynamic Semantics

{*elaboration (record_type_definition)* [partial]} The elaboration of a record_type_definition creates the record type and its first subtype, and consists of the elaboration of the record_definition. {*elaboration (record_definition)* [partial]} The elaboration of a record_definition consists of the elaboration of its component_list, if any. 16

{*elaboration (component_list)* [partial]} The elaboration of a component_list consists of the elaboration of the component_items and variant_part, if any, in the order in which they appear. {*elaboration (component_declaration)* [partial]} The elaboration of a component_declaration consists of the elaboration of the component_definition. 17

Discussion: If the defining_identifier_list has more than one defining_identifier, we presume here that the transformation explained in 3.3.1 has already taken place. Alternatively, we could say that the component_definition is elaborated once for each defining_identifier in the list. 17.a

{*8652/0002*} {*AI95-00171-01*} {*AI95-00230-01*} {*per-object expression*} {*per-object constraint*} {*entry index subtype*} Within the definition of a composite type, if a component_definition or discrete_subtype_definition (see 9.5.2) includes a name that denotes a discriminant of the type, or that is an attribute_reference whose prefix denotes the current instance of the type, the expression containing the name is called a *per-object expression*, and the constraint or range being defined is called a *per-object constraint*. {*elaboration (component_definition)* [partial]} For the elaboration of a component_definition of a component_declaration or the discrete_subtype_definition of an entry_-declaration for an entry family (see 9.5.2), if the component subtype is defined by an access_definition or if the constraint or range of the subtype_indication or discrete_subtype_definition is not a per-object 18/2 constraint, then the access_definition, subtype_indication, or discrete_subtype_definition is elaborated. On the other hand, if the constraint or range is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant.

18.1/1 {*8652/0002*} {*AI95-00171-01*} {*Elaboration (per-object constraint)* [partial]} When a per-object constraint is elaborated [(as part of creating an object)], each per-object expression of the constraint is evaluated. For other expressions, the values determined during the elaboration of the component_definition or entry_-declaration are used. Any checks associated with the enclosing subtype_indication or discrete_subtype_definition are performed[, including the subtype compatibility check (see 3.2.2),] and the associated subtype is created.

18.a **Discussion:** The evaluation of other expressions that appear in component_definitions and discrete_subtype_definitions is performed when the type definition is elaborated. The evaluation of expressions that appear as default_expressions is postponed until an object is created. Expressions in representation items that appear within a composite type definition are evaluated according to the rules of the particular representation item.

NOTES

19 57 A component_declaration with several identifiers is equivalent to a sequence of single component_declarations, as explained in 3.3.1.

20 58 The default_expression of a record component is only evaluated upon the creation of a default-initialized object of the record type (presuming the object has the component, if it is in a variant_part — see 3.3.1).

21 59 The subtype defined by a component_definition (see 3.6) has to be a definite subtype.

22 60 If a record type does not have a variant_part, then the same components are present in all values of the type.

23 61 A record type is limited if it has the reserved word **limited** in its definition, or if any of its components are limited (see 7.5).

24 62 {*predefined operations (of a record type)* [partial]} The predefined operations of a record type include membership tests, qualification, and explicit conversion. If the record type is nonlimited, they also include assignment and the predefined equality operators.

25/2 63 {*AI95-00287-01*} A component of a record can be named with a selected_component. A value of a record can be specified with a record_aggregate.

Examples

26 *Examples of record type declarations:*

27
```
type Date is
   record
      Day   : Integer range 1 .. 31;
      Month : Month_Name;
      Year  : Integer range 0 .. 4000;
   end record;
```

28
```
type Complex is
   record
      Re : Real := 0.0;
      Im : Real := 0.0;
   end record;
```

29 *Examples of record variables:*

30
```
Tomorrow, Yesterday : Date;
A, B, C : Complex;
```

31 -- *both components of A, B, and C are implicitly initialized to zero*

Extensions to Ada 83

31.a {*extensions to Ada 83*} The syntax rule for component_declaration is modified to use component_definition (instead of component_subtype_definition). The effect of this change is to allow the reserved word **aliased** before the component_subtype_definition.

A short-hand is provided for defining a null record type (and a null record extension), as these will be more common for abstract root types (and derived types without additional components). 31.b

The syntax rule for record_type_definition is modified to allow the reserved words **tagged** and **limited**. Tagging is new. Limitedness is now orthogonal to privateness. In Ada 83 the syntax implied that limited private was sort of more private than private. However, limitedness really has nothing to do with privateness; limitedness simply indicates the lack of assignment capabilities, and makes perfect sense for nonprivate types such as record types. 31.c

Wording Changes from Ada 83

{*8652/0009*} {*AI95-00137-01*} The syntax rules now allow aspect_clauses to appear in a record_definition. This is not a language extension, because Legality Rules prevent all language-defined representation clauses from appearing there. However, an implementation-defined attribute_definition_clause could appear there. The reason for this change is to allow the rules for aspect_clauses and representation pragmas to be as similar as possible. 31.d/1

Extensions to Ada 95

{*AI95-00287-01*} {*extensions to Ada 95*} Record components can have an anonymous access type. 31.e/2

{*AI95-00287-01*} {*extensions to Ada 95*} Limited components can be initialized, so long as the expression is one that allows building the object in place (such as an aggregate or function_call). 31.f/2

Wording Changes from Ada 95

{*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Improved the description of the elaboration of per-object constraints. 31.g/2

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Changed representation clauses to aspect clauses to reflect that they are used for more than just representation. 31.h/2

{*AI95-00318-02*} Defined *explicitly limited record* type to use in other rules. 31.i/2

3.8.1 Variant Parts and Discrete Choices

A record type with a variant_part specifies alternative lists of components. Each variant defines the components for the value or values of the discriminant covered by its discrete_choice_list. 1

Discussion: {*cover a value* [distributed]} Discrete_choice_lists and discrete_choices are said to *cover* values as defined below; which discrete_choice_list covers a value determines which of various alternatives is chosen. These are used in variant_parts, array_aggregates, and case_statements. 1.a

Language Design Principles

The definition of “cover” in this subclause and the rules about discrete choices are designed so that they are also appropriate for array aggregates and case statements. 1.b

The rules of this subclause intentionally parallel those for case statements. 1.c

Syntax

2
```
variant_part ::=
  case discriminant_direct_name is
     variant
    {variant}
  end case;
```

3
```
variant ::=
  when discrete_choice_list =>
    component_list
```

4
```
discrete_choice_list ::= discrete_choice {| discrete_choice}
```

5
```
discrete_choice ::= expression | discrete_range | others
```

Name Resolution Rules

{*discriminant (of a variant_part)*} The *discriminant*_direct_name shall resolve to denote a discriminant (called the *discriminant of the variant_part*) specified in the known_discriminant_part of the 6 full_type_declaration that contains the variant_part. {*expected type (variant_part discrete_choice)* [partial]} The expected type for each discrete_choice in a variant is the type of the discriminant of the variant_part.

6.a **Ramification:** A full_type_declaration with a variant_part has to have a (new) known_discriminant_part; the discriminant of the variant_part cannot be an inherited discriminant.

Legality Rules

7 The discriminant of the variant_part shall be of a discrete type.

7.a **Ramification:** It shall not be of an access type, named or anonymous.

8 The expressions and discrete_ranges given as discrete_choices in a variant_part shall be static. The discrete_choice **others** shall appear alone in a discrete_choice_list, and such a discrete_choice_list, if it appears, shall be the last one in the enclosing construct.

9 {*cover a value (by a discrete_choice)* [partial]} A discrete_choice is defined to *cover a value* in the following cases:

10 • A discrete_choice that is an expression covers a value if the value equals the value of the expression converted to the expected type.

11 • A discrete_choice that is a discrete_range covers all values (possibly none) that belong to the range.

12 • The discrete_choice **others** covers all values of its expected type that are not covered by previous discrete_choice_lists of the same construct.

12.a **Ramification:** For case_statements, this includes values outside the range of the static subtype (if any) to be covered by the choices. It even includes values outside the base range of the case expression's type, since values of numeric types (and undefined values of any scalar type?) can be outside their base range.

13 {*cover a value (by a discrete_choice_list)* [partial]} A discrete_choice_list covers a value if one of its discrete_choices covers the value.

14 The possible values of the discriminant of a variant_part shall be covered as follows:

15 • If the discriminant is of a static constrained scalar subtype, then each non-**others** discrete_choice shall cover only values in that subtype, and each value of that subtype shall be covered by some discrete_choice [(either explicitly or by **others**)];

16 • If the type of the discriminant is a descendant of a generic formal scalar type then the variant_part shall have an **others** discrete_choice;

16.a **Reason:** The base range is not known statically in this case.

17 • Otherwise, each value of the base range of the type of the discriminant shall be covered [(either explicitly or by **others**)].

18 Two distinct discrete_choices of a variant_part shall not cover the same value.

Static Semantics

19 If the component_list of a variant is specified by **null**, the variant has no components.

20 {*govern a variant_part*} {*govern a variant*} The discriminant of a variant_part is said to *govern* the variant_part and its variants. In addition, the discriminant of a derived type governs a variant_part and its variants if it corresponds (see 3.7) to the discriminant of the variant_part.

Dynamic Semantics

21 A record value contains the values of the components of a particular variant only if the value of the discriminant governing the variant is covered by the discrete_choice_list of the variant. This rule applies in turn to any further variant that is, itself, included in the component_list of the given variant.

22 {*elaboration (variant_part)* [partial]} The elaboration of a variant_part consists of the elaboration of the component_list of each variant in the order in which they appear.

Examples

23 *Example of record type with a variant part:*

24
```
type Device is (Printer, Disk, Drum);
type State  is (Open, Closed);
```

25
```
type Peripheral(Unit : Device := Disk) is
   record
      Status : State;
      case Unit is
         when Printer =>
            Line_Count : Integer range 1 .. Page_Size;
         when others =>
            Cylinder   : Cylinder_Index;
            Track      : Track_Number;
         end case;
      end record;
```

26 *Examples of record subtypes:*

27
```
subtype Drum_Unit is Peripheral(Drum);
subtype Disk_Unit is Peripheral(Disk);
```

28 *Examples of constrained record variables:*

29
```
Writer   : Peripheral(Unit  => Printer);
Archive  : Disk_Unit;
```

Extensions to Ada 83

29.a {*extensions to Ada 83*} In Ada 83, the discriminant of a variant_part is not allowed to be of a generic formal type. This restriction is removed in Ada 95; an **others** discrete_choice is required in this case.

Wording Changes from Ada 83

29.b The syntactic category choice is removed. The syntax rules for variant, array_aggregate, and case_statement now use discrete_choice_list or discrete_choice instead. The syntax rule for record_aggregate now defines its own syntax for named associations.

29.c We have added the term Discrete Choice to the title since this is where they are talked about. This is analogous to the name of the subclause "Index Constraints and Discrete Ranges" in the clause on Array Types.

29.d The rule requiring that the discriminant denote a discriminant of the type being defined seems to have been left implicit in RM83.

3.9 Tagged Types and Type Extensions

1 [{*dispatching operation* [partial]} {*polymorphism*} {*dynamic binding: See dispatching operation*} {*generic unit: See also dispatching operation*} {*variant: See also tagged type*} Tagged types and type extensions support object-oriented programming, based on inheritance with extension and run-time polymorphism via *dispatching operations*. {*object-oriented programming (OOP): See tagged types and type extensions*} {*OOP (object-oriented programming): See tagged types and type extensions*} {*inheritance: See also tagged types and type extension*}]

Language Design Principles

1.a/2 {*AI95-00251-01*} The intended implementation model is for the static portion of a tag to be represented as a pointer to a statically allocated and link-time initialized type descriptor. The type descriptor contains the address of the code for each primitive operation of the type. It probably also contains other information, such as might make membership tests convenient and efficient. Tags for nested type extensions must also have a dynamic part that identifies the particular elaboration of the type.

1.b The primitive operations of a tagged type are known at its first freezing point; the type descriptor is laid out at that point. It contains linker symbols for each primitive operation; the linker fills in the actual addresses.

1.b.1/2 {*AI95-00251-01*} Primitive operations of type extensions that are declared at a level deeper than the level of the ultimate ancestor from which they are derived can be represented by wrappers that use the dynamic part of the tag to call the actual primitive operation. The dynamic part would generally be some way to represent the static link or display necessary for making a nested call. One implementation strategy would be to store that information in the extension part of such nested type extensions, and use the dynamic part of the tag to point at it. (That way, the "dynamic" part of the tag could be static, at the cost of indirect access.)

1.b.2/2 {*AI95-00251-01*} If the tagged type is descended from any interface types, it also will need to include "subtags" (one for each interface) that describe the mapping of the primitive operations of the interface to the primitives of the type. These subtags could directly reference the primitive operations (for faster performance), or simply provide the tag "slot" numbers for the primitive operations (for easier derivation). In either case, the subtags would be used for calls that dispatch through a class-wide type of the interface.

1.c Other implementation models are possible.

1.d The rules ensure that "dangling dispatching" is impossible; that is, when a dispatching call is made, there is always a body to execute. This is different from some other object-oriented languages, such as Smalltalk, where it is possible to get a run-time error from a missing method.

1.e/2 {*AI95-00251-01*} Dispatching calls should be efficient, and should have a bounded worst-case execution time. This is important in a language intended for real-time applications. In the intended implementation model, a dispatching call involves calling indirect through the appropriate slot in the dispatch table. No complicated "method lookup" is involved although a call which is dispatching on an interface may require a lookup of the appropriate interface subtag.

1.f The programmer should have the choice at each call site of a dispatching operation whether to do a dispatching call or a statically determined call (i.e. whether the body executed should be determined at run time or at compile time).

1.g The same body should be executed for a call where the tag is statically determined to be T'Tag as for a dispatching call where the tag is found at run time to be T'Tag. This allows one to test a given tagged type with statically determined calls, with some confidence that run-time dispatching will produce the same behavior.

1.h All views of a type should share the same type descriptor and the same tag.

1.i The visibility rules determine what is legal at compile time; they have nothing to do with what bodies can be executed at run time. Thus, it is possible to dispatch to a subprogram whose declaration is not visible at the call site. In fact, this is one of the primary facts that gives object-oriented programming its power. The subprogram that ends up being dispatched to by a given call might even be designed long after the call site has been coded and compiled.

1.j Given that Ada has overloading, determining whether a given subprogram overrides another is based both on the names and the type profiles of the operations.

1.k/2 {*AI95-00401-01*} When a type extension is declared, if there is any place within its immediate scope where a certain subprogram of the parent or progenitor is visible, then a matching subprogram should override. If there is no such place, then a matching subprogram should be totally unrelated, and occupy a different slot in the type descriptor. This is important to preserve the privacy of private parts; when an operation declared in a private part is inherited, the inherited version can be overridden only in that private part, in the package body, and in any children of the package.

1.l If an implementation shares code for instances of generic bodies, it should be allowed to share type descriptors of tagged types declared in the generic body, so long as they are not extensions of types declared in the specification of the generic unit.

Static Semantics

2/2 {*AI95-00345-01*} {*tagged type*} A record type or private type that has the reserved word **tagged** in its declaration is called a *tagged* type. In addition, an interface type is a tagged type, as is a task or protected type derived from an interface (see 3.9.4). [When deriving from a tagged type, as for any derived type, additional primitive subprograms may be defined, and inherited primitive subprograms may be overridden.] {*type extension*} {*extension (of a type)*} The derived type is called an *extension* of its ancestor types, or simply a *type extension*.

2.1/2 {*AI95-00345-01*} {*extension (of a record type)*} {*private extension*} {*extension (of a private type)*} Every type extension is also a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a non-interface synchronized tagged type (see 3.9.4). A record extension is defined by a derived_type_definition with a record_extension_part (see 3.9.1)[, which may include the definition of additional components]. A private extension, which is a partial view of a record extension or of a synchronized tagged type, can be declared in the visible part of a package (see 7.3) or in a generic formal part (see 12.5.1).

2.a **Glossary entry:** {*Tagged type*} The objects of a tagged type have a run-time type tag, which indicates the specific type with which the object was originally created. An operand of a class-wide tagged type can be used in a dispatching call; the tag indicates which subprogram body to invoke. Nondispatching calls, in which the subprogram body to invoke is determined at compile time, are also allowed. Tagged types may be extended with additional components.

2.b/2 **Ramification:** {*AI95-00218-03*} If a tagged type is declared other than in a package_specification, it is impossible to add new primitive subprograms for that type, although it can inherit primitive subprograms, and those can be overridden. If the user incorrectly thinks a certain subprogram is primitive when it is not, and tries to call it with a dispatching call, an error message will be given at the call site. Similarly, by using an overriding_indicator (see 6.1), the user can declare that a subprogram is intended to be overriding, and get an error message when they made a mistake. The use of overriding_indicators is highly recommended in new code that does not need to be compatible with Ada 95.

3 {*tag of an object*} An object of a tagged type has an associated (run-time) *tag* that identifies the specific tagged type used to create the object originally. [The tag of an operand of a class-wide tagged type *T*'Class controls which subprogram body is to be executed when a primitive subprogram of type *T* is applied to the operand (see 3.9.2); {*dispatching*} using a tag to control which body to execute is called *dispatching*.] {*type tag: See tag*} {*run-time type: See tag*} {*type: See also tag*} {*class: See also tag*}

4/2 {*AI95-00344-01*} The tag of a specific tagged type identifies the full_type_declaration of the type, and for a type extension, is sufficient to uniquely identify the type among all descendants of the same ancestor. If a declaration for a tagged type occurs within a generic_package_declaration, then the corresponding type declarations in distinct instances of the generic package are associated with distinct tags. For a tagged type that is local to a generic package body and with all of its ancestors (if any) also local to the generic body, the language does not specify whether repeated instantiations of the generic body result in distinct tags.{*Unspecified* [partial]}

4.a/2 *This paragraph was deleted.*{*AI95-00344-01*}

4.a.1/2 **Implementation Note:** {*AI95-00344-01*} In most cases, a tag need only identify a particular tagged type declaration, and can therefore be a simple link-time-known address. However, for tag checks (see 3.9.2) it is essential that each descendant (that currently exists) of a given type have a unique tag. Hence, for types declared in shared generic bodies where an ancestor comes from outside the generic, or for types declared at a deeper level than an ancestor, the tag needs to be augmented with some kind of dynamic descriptor (which may be a static link, global display, instance descriptor pointer, or combination). This implies that type Tag may need to be two words, the second of which is normally null, but in these identified special cases needs to include a static link or equivalent. Within an object of one of these types with a two-word tag, the two parts of the tag would typically be separated, one part as the first word of the object, the second placed in the first extension part that corresponds to a type declared more nested than its parent or declared in a shared generic body when the parent is declared outside. Alternatively, by using an extra level of indirection, the type Tag could remain a single-word.

4.b/2 {*AI95-00344-01*} For types that are not type extensions (even for ones declared in nested scopes), we do not require that repeated elaborations of the same full_type_declaration correspond to distinct tags. This was done so that Ada 2005 implementations of tagged types could maintain representation compatibility with Ada 95 implementations. Only type extensions that were not allowed in Ada 95 require additional information with the tag.

4.c/2 **To be honest:** {*AI95-00344-01*} The wording "is sufficient to uniquely identify the type among all descendants of the same ancestor" only applies to types that currently exist. It is not necessary to distinguish between descendants that currently exist, and descendants of the same type that no longer exist. For instance, the address of the stack frame of the subprogram that created the tag is sufficient to meet the requirements of this rule, even though it is possible, after the subprogram returns, that a later call of the subprogram could have the same stack frame and thus have an identical tag.

5 The following language-defined library package exists:

```
6/2      {AI95-00362-01} package Ada.Tags is
             pragma Preelaborate(Tags);
             type Tag is private;
             pragma Preelaborable_Initialization(Tag);

6.1/2    {AI95-00260-02}     No_Tag : constant Tag;

7/2      {AI95-00400-01}     function Expanded_Name(T : Tag) return String;
             function Wide_Expanded_Name(T : Tag) return Wide_String;
             function Wide_Wide_Expanded_Name(T : Tag) return Wide_Wide_String;
             function External_Tag(T : Tag) return String;
             function Internal_Tag(External : String) return Tag;

7.1/2    {AI95-00344-01}     function Descendant_Tag(External : String; Ancestor : Tag)
         return Tag;
             function Is_Descendant_At_Same_Level(Descendant, Ancestor : Tag)
                 return Boolean;

7.2/2    {AI95-00260-02}     function Parent_Tag (T : Tag) return Tag;

7.3/2    {AI95-00405-01}     type Tag_Array is array (Positive range <>) of Tag;

7.4/2    {AI95-00405-01}     function Interface_Ancestor_Tags (T : Tag) return Tag_Array;

8            Tag_Error : exception;

9        private
            ... -- not specified by the language
         end Ada.Tags;
```

9.a **Reason:** Tag is a nonlimited, definite subtype, because it needs the equality operators, so that tag checking makes sense. Also, equality, assignment, and object declaration are all useful capabilities for this subtype.

9.b For an object X and a type T, "X'Tag = T'Tag" is not needed, because a membership test can be used. However, comparing the tags of two objects cannot be done via membership. This is one reason to allow equality for type Tag.

9.1/2 {*AI95-00260-02*} No_Tag is the default initial value of type Tag.

9.c/2 **Reason:** {*AI95-00260-02*} This is similar to the requirement that all access values be initialized to **null**.

10/2 {*AI95-00400-01*} The function Wide_Wide_Expanded_Name returns the full expanded name of the first subtype of the specific type identified by the tag, in upper case, starting with a root library unit. The result is implementation defined if the type is declared within an unnamed block_statement.

10.a **To be honest:** This name, as well as each prefix of it, does not denote a renaming_declaration.

10.b/2 **Implementation defined:** The result of Tags.Wide_Wide_Expanded_Name for types declared within an unnamed block_statement.

10.1/2 {*AI95-00400-01*} The function Expanded_Name (respectively, Wide_Expanded_Name) returns the same sequence of graphic characters as that defined for Wide_Wide_Expanded_Name, if all the graphic characters are defined in Character (respectively, Wide_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide_Wide_Expanded_Name for the same value of the argument.

10.c/2 **Implementation defined:** The sequence of characters of the value returned by Tags.Expanded_Name (respectively, Tags.Wide_Expanded_Name) when some of the graphic characters of Tags.Wide_Wide_Expanded_Name are not defined in Character (respectively, Wide_Character).

11 The function External_Tag returns a string to be used in an external representation for the given tag. The call External_Tag(S'Tag) is equivalent to the attribute_reference S'External_Tag (see 13.3).

11.a **Reason:** It might seem redundant to provide both the function External_Tag and the attribute External_Tag. The function is needed because the attribute can't be applied to values of type Tag. The attribute is needed so that it can be specified via an attribute_definition_clause.

11.1/2 {*AI95-00417-01*} The string returned by the functions Expanded_Name, Wide_Expanded_Name, Wide_Wide_Expanded_Name, and External_Tag has lower bound 1.

{*AI95-00279-01*} The function Internal_Tag returns a tag that corresponds to the given external tag, or raises Tag_Error if the given string is not the external tag for any specific type of the partition. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created (see 13.14). 12/2

Reason: {*AI95-00279-01*} The check for uncreated library-level types prevents a reference to the type before execution reaches the freezing point of the type. This is important so that T'Class'Input or an instance of Tags.Generic_Dispatching_Constructor do not try to create an object of a type that hasn't been frozen (which may not have yet elaborated its constraints). We don't require this behavior for non-library-level types as the tag can be created multiple times and possibly multiple copies can exist at the same time, making the check complex. 12.a/2

{*AI95-00344-01*} The function Descendant_Tag returns the (internal) tag for the type that corresponds to the given external tag and is both a descendant of the type identified by the Ancestor tag and has the same accessibility level as the identified ancestor. Tag_Error is raised if External is not the external tag for such a type. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created. 12.1/2

Reason: Descendant_Tag is used by T'Class'Input to identify the type identified by an external tag. Because there can be multiple elaborations of a given type declaration, Internal_Tag does not have enough information to choose a unique such type. Descendant_Tag does not return the tag for types declared at deeper accessibility levels than the ancestor because there could be ambiguity in the presence of recursion or multiple tasks. Descendant_Tag can be used in constructing a user-defined replacement for T'Class'Input. 12.b/2

{*AI95-00344-01*} The function Is_Descendant_At_Same_Level returns True if the Descendant tag identifies a type that is both a descendant of the type identified by Ancestor and at the same accessibility level. If not, it returns False. 12.2/2

Reason: Is_Descendant_At_Same_Level (or something similar to it) is used by T'Class'Output to determine whether the item being written is at the same accessibility level as T. It may be used to determine prior to using T'Class'Output whether Tag_Error will be raised, and also can be used in constructing a user-defined replacement for T'Class'Output. 12.c/2

{*AI95-00260-02*} The function Parent_Tag returns the tag of the parent type of the type whose tag is T. If the type does not have a parent type (that is, it was not declared by a derived_type_declaration), then No_Tag is returned. 12.3/2

Ramification: The parent type is always the parent of the full type; a private extension appears to define a parent type, but it does not (only the various forms of derivation do that). As this is a run-time operation, ignoring privateness is OK. 12.d/2

{*AI95-00405-01*} The function Interface_Ancestor_Tags returns an array containing the tag of each interface ancestor type of the type whose tag is T, other than T itself. The lower bound of the returned array is 1, and the order of the returned tags is unspecified. Each tag appears in the result exactly once.[If the type whose tag is T has no interface ancestors, a null array is returned.]{*Unspecified* [partial]} 12.4/2

Ramification: The result of Interface_Ancestor_Tags includes the tag of the parent type, if the parent is an interface. 12.e/2

Indirect interface ancestors are included in the result of Interface_Ancestor_Tags. That's because where an interface appears in the derivation tree has no effect on the semantics of the type; the only interesting property is whether the type has an interface as an ancestor. 12.f/2

For every subtype S of a tagged type *T* (specific or class-wide), the following attributes are defined: 13

S'Class S'Class denotes a subtype of the class-wide type (called *T*'Class in this International Standard) for the class rooted at *T* (or if S already denotes a class-wide subtype, then S'Class is the same as S). 14

{*unconstrained (subtype)*} {*constrained (subtype)*} S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type *T* belong to S. 15

Ramification: This attribute is defined for both specific and class-wide subtypes. The definition is such that S'Class'Class is the same as S'Class. 15.a

15.b Note that if S is constrained, S'Class is only partially constrained, since there might be additional discriminants added in descendants of *T* which are not constrained.

15.c/2 **Reason:** {*AI95-00326-01*} The Class attribute is not defined for untagged subtypes (except for incomplete types and private types whose full view is tagged — see J.11 and 7.3.1) so as to preclude implicit conversion in the absence of run-time type information. If it were defined for untagged subtypes, it would correspond to the concept of universal types provided for the predefined numeric classes.

16 S'Tag S'Tag denotes the tag of the type *T* (or if *T* is class-wide, the tag of the root type of the corresponding class). The value of this attribute is of type Tag.

16.a **Reason:** S'Class'Tag equals S'Tag, to avoid generic contract model problems when S'Class is the actual type associated with a generic formal derived type.

17 Given a prefix X that is of a class-wide tagged type [(after any implicit dereference)], the following attribute is defined:

18 X'Tag X'Tag denotes the tag of X. The value of this attribute is of type Tag.

18.a **Reason:** X'Tag is not defined if X is of a specific type. This is primarily to avoid confusion that might result about whether the Tag attribute should reflect the tag of the type of X, or the tag of X. No such confusion is possible if X is of a class-wide type.

18.1/2 {*AI95-00260-02*} {*AI95-00441-01*} The following language-defined generic function exists:

18.2/2
```
generic
    type T (<>) is abstract tagged limited private;
    type Parameters (<>) is limited private;
    with function Constructor (Params : not null access Parameters)
        return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
   (The_Tag : Tag;
    Params  : not null access Parameters) return T'Class;
pragma Preelaborate(Generic_Dispatching_Constructor);
pragma Convention(Intrinsic, Generic_Dispatching_Constructor);
```

18.3/2 {*AI95-00260-02*} Tags.Generic_Dispatching_Constructor provides a mechanism to create an object of an appropriate type from just a tag value. The function Constructor is expected to create the object given a reference to an object of type Parameters.

18.b/2 **Discussion:** This specification is designed to make it easy to create dispatching constructors for streams; in particular, this can be used to construct overridings for T'Class'Input.

18.c/2 Note that any tagged type will match T (see 12.5.1).

Dynamic Semantics

19 The tag associated with an object of a tagged type is determined as follows:

20 • {*tag of an object (stand-alone object, component, or aggregate)* [partial]} The tag of a stand-alone object, a component, or an aggregate of a specific tagged type *T* identifies *T*.

20.a **Discussion:** The tag of a formal parameter of type *T* is not necessarily the tag of *T*, if, for example, the actual was a type conversion.

21 • {*tag of an object (object created by an allocator)* [partial]} The tag of an object created by an allocator for an access type with a specific designated tagged type *T*, identifies *T*.

21.a **Discussion:** The tag of an object designated by a value of such an access type might not be *T*, if, for example, the access value is the result of a type conversion.

22 • {*tag of an object (class-wide object)* [partial]} The tag of an object of a class-wide tagged type is that of its initialization expression.

22.a **Ramification:** The tag of an object (even a class-wide one) cannot be changed after it is initialized, since a "class-wide" assignment_statement raises Constraint_Error if the tags don't match, and a "specific" assignment_statement does not affect the tag.

23 • {*tag of an object (returned by a function)* [partial]} The tag of the result returned by a function whose result type is a specific tagged type *T* identifies *T*.

Implementation Note: {*AI95-00318-02*} For a limited tagged type, the return object is "built in place" in the ultimate result object with the appropriate tag. For a nonlimited type, a new anonymous object with the appropriate tag is created as part of the function return. See 6.5, "Return Statements". 23.a/2

- {*AI95-00318-02*} {*tag of an object (returned by a function)* [partial]} The tag of the result returned by a function with a class-wide result type is that of the return object. 24/2

{*tag of an object (preserved by type conversion and parameter passing)* [partial]} The tag is preserved by type conversion and by parameter passing. The tag of a value is the tag of the associated object (see 6.2). 25

{*AI95-00260-02*} {*AI95-00344-01*} {*AI95-00405-01*} Tag_Error is raised by a call of Descendant_Tag, Expanded_Name, External_Tag, Interface_Ancestor_Tag, Is_Descendant_At_Same_Level, or Parent_Tag if any tag passed is No_Tag. 25.1/2

{*AI95-00260-02*} An instance of Tags.Generic_Dispatching_Constructor raises Tag_Error if The_Tag does not represent a concrete descendant of T or if the innermost master (see 7.6.1) of this descendant is not also a master of the instance. Otherwise, it dispatches to the primitive function denoted by the formal Constructor for the type identified by The_Tag, passing Params, and returns the result. Any exception raised by the function is propagated. 25.2/2

Ramification: The tag check checks both that The_Tag is in T'Class, and that it is not abstract. These checks are similar to the ones required by streams for T'Class'Input (see 13.13.2). In addition, there is a check that the tag identifies a type declared on the current dynamic call chain, and not a more nested type or a type declared by another task. This check is not necessary for streams, because the stream attributes are declared at the same dynamic level as the type used. 25.a/2

Erroneous Execution

{*AI95-00260-02*} {*erroneous execution (cause)* [partial]} If an internal tag provided to an instance of Tags.Generic_Dispatching_Constructor or to any subprogram declared in package Tags identifies either a type that is not library-level and whose tag has not been created (see 13.14), or a type that does not exist in the partition at the time of the call, then execution is erroneous. 25.3/2

Ramification: One reason that a type might not exist in the partition is that the tag refers to a type whose declaration was elaborated as part of an execution of a subprogram_body which has been left (see 7.6.1). 25.b/2

We exclude tags of library-level types from the current execution of the partition, because misuse of such tags should always be detected. T'Tag freezes the type (and thus creates the tag), and Internal_Tag and Descendant_Tag cannot return the tag of a library-level type that has not been created. All ancestors of a tagged type must be frozen no later than the (full) declaration of a type that uses them, so Parent_Tag and Interface_Ancestor_Tags cannot return a tag that has not been created. Finally, library-level types never cease to exist while the partition is executing. Thus, if the tag comes from a library-level type, there cannot be erroneous execution (the use of Descendant_Tag rather than Internal_Tag can help ensure that the tag is of a library-level type). This is also similar to the rules for T'Class'Input (see 13.13.2). 25.c/2

Discussion: {*AI95-00344-01*} Ada 95 allowed Tag_Error in this case, or expected the functions to work. This worked because most implementations used tags constructed at link-time, and each elaboration of the same type_declaration produced the same tag. However, Ada 2005 requires at least part of the tags to be dynamically constructed for a type derived from a type at a shallower level. For dynamically constructed tags, detecting the error can be expensive and unreliable. To see this, consider a program containing two tasks. Task A creates a nested tagged type, passes the tag to task B (which saves it), and then terminates. The nested tag (if dynamic) probably will need to refer in some way to the stack frame for task A. If task B later tries to use the tag created by task A, the tag's reference to the stack frame of A probably is a dangling pointer. Avoiding this would require some sort of protected tag manager, which would be a bottleneck in a program's performance. Moreover, we'd still have a race condition; if task A terminated after the tag check, but before the tag was used, we'd still have a problem. That means that all of these operations would have to be serialized. That could be a significant performance drain, whether or not nested tagged types are every used. Therefore, we allow execution to become erroneous as we do for other dangling pointers. If the implementation can detect the error, we recommend that Tag_Error be raised. 25.d/2

Implementation Permissions

{*AI95-00260-02*} {*AI95-00279-01*} The implementation of Internal_Tag and Descendant_Tag may raise Tag_Error if no specific type corresponding to the string External passed as a parameter exists in the 26/2 partition at the time the function is called, or if there is no such type whose innermost master is a master of the point of the function call.

26.a/2 **Reason:** {*AI95-00260-02*} {*AI95-00279-01*} {*AI95-00344-01*} Locking would be required to ensure that the mapping of strings to tags never returned tags of types which no longer exist, because types can cease to exist (because they belong to another task, as described above) during the execution of these operations. Moreover, even if these functions did use locking, that would not prevent the type from ceasing to exist at the instant that the function returned. Thus, we do not require the overhead of locking; hence the word “may” in this rule.

Implementation Advice

26.1/2 {*AI95-00260-02*} Internal_Tag should return the tag of a type whose innermost master is the master of the point of the function call.

26.b/2 **Implementation Advice:** Tags.Internal_Tag should return the tag of a type whose innermost master is the master of the point of the function call..

26.c/2 **Reason:** {*AI95-00260-02*} {*AI95-00344-01*} It's not helpful if Internal_Tag returns the tag of some type in another task when one is available in the task that made the call. We don't require this behavior (because it requires the same implementation techniques we decided not to insist on previously), but encourage it.

NOTES

27 64 A type declared with the reserved word **tagged** should normally be declared in a package_specification, so that new primitive subprograms can be declared for it.

28 65 Once an object has been created, its tag never changes.

29 66 Class-wide types are defined to have unknown discriminants (see 3.7). This means that objects of a class-wide type have to be explicitly initialized (whether created by an object_declaration or an allocator), and that aggregates have to be explicitly qualified with a specific type when their expected type is class-wide.

30/2 *This paragraph was deleted.*{*AI95-00326-01*}

30.1/2 67 {*AI95-00260-02*} The capability provided by Tags.Generic_Dispatching_Constructor is sometimes known as a *factory*.{*factory*} {*class factory*}

Examples

31 *Examples of tagged record types:*

32
```
type Point is tagged
  record
    X, Y : Real := 0.0;
  end record;
```

33
```
type Expression is tagged null record;
  -- Components will be added by each extension
```

Extensions to Ada 83

33.a {*extensions to Ada 83*} Tagged types are a new concept.

Inconsistencies With Ada 95

33.b/2 {*AI95-00279-01*} {*inconsistencies with Ada 95*} **Amendment Correction:** Added wording specifying that Internal_Tag must raise Tag_Error if the tag of a library-level type has not yet been created. Ada 95 gave an Implementation Permission to do this; we require it to avoid erroneous execution when streaming in an object of a library-level type that has not yet been elaborated. This is technically inconsistent; a program that used Internal_Tag outside of streaming and used a compiler that didn't take advantage of the Implementation Permission would not have raised Tag_Error, and may have returned a useful tag. (If the tag was used in streaming, the program would have been erroneous.) Since such a program would not have been portable to a compiler that did take advantage of the Implementation Permission, this is not a significant inconsistency.

33.c/2 {*AI95-00417-01*} We now define the lower bound of the string returned from [[Wide_]Wide_]Expanded_Name and External_Name. This makes working with the returned string easier, and is consistent with many other string-returning functions in Ada. This is technically an inconsistency; if a program depended on some other lower bound for the string returned from one of these functions, it could fail when compiled with Ada 2005. Such code is not portable even between Ada 95 implementations, so it should be very rare.

Incompatibilities With Ada 95

{*AI95-00260-02*} {*AI95-00344-01*} {*AI95-00400-01*} {*AI95-00405-01*} {*incompatibilities with Ada 95*} Constant No_Tag, and functions Parent_Tag, Interface_Ancestor_Tags, Descendant_Tag, Is_Descendant_At_Same_Level, Wide_Expanded_Name, and Wide_Wide_Expanded_Name are newly added to Ada.Tags. If Ada.Tags is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Ada.Tags is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 33.d/2

Extensions to Ada 95

{*AI95-00362-01*} {*extensions to Ada 95*} Ada.Tags is now defined to be preelaborated. 33.e/2

{*AI95-00260-02*} Generic function Tags.Generic_Dispatching_Constructor is new. 33.f/2

Wording Changes from Ada 95

{*AI95-00318-02*} We talk about return objects rather than return expressions, as functions can return using an extended_return_statement. 33.g/2

{*AI95-00344-01*} Added wording to define that tags for all descendants of a tagged type must be distinct. This is needed to ensure that more nested type extensions will work properly. The wording does not require implementation changes for types that were allowed in Ada 95. 33.h/2

3.9.1 Type Extensions

{*AI95-00345-01*} [{*type extension*} {*extension (of a type)*} {*record extension*} {*extension (of a record type)*} {*private extension*} {*extension (of a private type)*} Every type extension is a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a non-interface synchronized tagged type..] 1/2

Language Design Principles

We want to make sure that we can extend a generic formal tagged type, without knowing its discriminants. 1.a

We don't want to allow components in an extension aggregate to depend on discriminants inherited from the parent value, since such dependence requires staticness in aggregates, at least for variants. 1.b

Syntax

record_extension_part ::= **with** record_definition 2

Legality Rules

{*AI95-00344-01*} {*AI95-00345-01*} {*AI95-00419-01*} The parent type of a record extension shall not be a class-wide type nor shall it be a synchronized tagged type (see 3.9.4). If the parent type or any progenitor is nonlimited, then each of the components of the record_extension_part shall be nonlimited. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 3/2

Reason: If the parent is a limited formal type, then the actual might be nonlimited. 3.a

{*AI95-00344-01*} Ada 95 required the record extensions to be the same level as the parent type. Now we use accessibility checks on class-wide allocators and return statements to prevent objects from living longer than their type. 3.b/2

{*AI95-00345-01*} Synchronized tagged types cannot be extended. We have this limitation so that all of the data of a task or protected type is defined within the type. Data defined outside of the type wouldn't be subject to the mutual exclusion properties of a protected type, and couldn't be used by a task, and thus doesn't seem to be worth the potential impact on implementations. 3.c/2

{*AI95-00344-01*} Within the body of a generic unit, or the body of any of its descendant library units, a tagged type shall not be declared as a descendant of a formal type declared within the formal part of the generic unit. 4/2

Reason: This paragraph ensures that a dispatching call will never attempt to execute an inaccessible subprogram body. 4.a

4.a.1/2 {*AI95-00344-01*} The convoluted wording ("formal type declared within the formal part") is necessary to include tagged types that are formal parameters of formal packages of the generic unit, as well as formal tagged and tagged formal derived types of the generic unit.

4.b/2 {*AI95-00344-01*} This rule is necessary in order to preserve the contract model.

4.c/2 {*AI95-00344-01*} If an ancestor is a formal of the generic unit , we have a problem because it might have an unknown number of abstract subprograms, as in the following example:

4.d/2
```
package P is
    type T is tagged null record;
    function F return T; -- Inherited versions will require overriding.
end P;
```

4.e
```
generic
    type TT is tagged private;
package Gp is
    type NT is abstract new TT with null record;
    procedure Q(X : in NT) is abstract;
end Gp;
```

4.f/2
```
package body Gp is
    type NT2 is new NT with null record; -- Illegal!
    procedure Q(X : in NT2) is begin null; end Q;
    -- Is this legal or not? Can't decide because
    -- we don't know whether TT had any functions that require
    -- overriding on extension.
end Gp;
```

4.g
```
package I is new Gp(TT => P.T);
```

4.h/2 I.NT is an abstract type with two abstract subprograms: F (inherited as abstract) and Q (explicitly declared as abstract). But the generic body doesn't know about F, so we don't know that it needs to be overridden to make a nonabstract extension of NT.Hence, we have to disallow this case.

4.h.1/2 Similarly, since the actual type for a formal tagged limited private type can be a nonlimited type, we would have a problem if a type extension of a limited private formal type could be declared in a generic body. Such an extension could have a task component, for example, and an object of that type could be passed to a dispatching operation of a nonlimited ancestor type. That operation could try to copy the object with the task component. That would be bad. So we disallow this as well.

4.i If TT were declared as abstract, then we could have the same problem with abstract procedures.

4.j We considered disallowing all tagged types in a generic body, for simplicity. We decided not to go that far, in order to avoid unnecessary restrictions.

4.k {*accessibility rule (not part of generic contract)* [partial]} We also considered trying make the accessibility level part of the contract; i.e. invent some way of saying (in the generic_declaration) "all instances of this generic unit will have the same accessibility level as the generic_declaration." Unfortunately, that doesn't solve the part of the problem having to do with abstract types.

4.l/2 *This paragraph was deleted.*

4.m/2 **Ramification:** {*AI95-00344*} This rule applies to types with ancestors (directly or indirectly) of formal interface types (see 12.5.5), formal tagged private types (see 12.5.1), and formal derived private types whose ancestor type is tagged (see 12.5.1).

Static Semantics

4.1/2 {*AI95-00391-01*} {*null extension*} A record extension is a *null extension* if its declaration has no known_discriminant_part and its record_extension_part includes no component_declarations.

Dynamic Semantics

5 {*elaboration (record_extension_part)* [partial]} The elaboration of a record_extension_part consists of the elaboration of the record_definition.

NOTES

6 68 The term "type extension" refers to a type as a whole. The term "extension part" refers to the piece of text that defines the additional components (if any) the type extension has relative to its specified ancestor type.

6.a **Discussion:** We considered other terminology, such as "extended type." However, the terms "private extended type" and "record extended type" did not convey the proper meaning. Hence, we have chosen to uniformly use the term "extension" as the type resulting from extending a type, with "private extension" being one produced by privately extending the type, and "record extension" being one produced by extending the type with an additional record-like set of components. Note also that the term "type extension" refers to the result of extending a type in the language Oberon as well (though there the term "extended type" is also used, interchangeably, perhaps because Oberon doesn't have the concept of a "private extension").

69 {*AI95-00344-01*} When an extension is declared immediately within a body, primitive subprograms are inherited and are overridable, but new primitive subprograms cannot be added. 7/2

70 A name that denotes a component (including a discriminant) of the parent type is not allowed within the record_extension_part. Similarly, a name that denotes a component defined within the record_extension_part is not allowed within the record_extension_part. It is permissible to use a name that denotes a discriminant of the record extension, providing there is a new known_discriminant_part in the enclosing type declaration. (The full rule is given in 3.8.) 8

Reason: The restriction against depending on discriminants of the parent is to simplify the definition of extension aggregates. The restriction against using parent components in other ways is methodological; it presumably simplifies implementation as well. 8.a

71 Each visible component of a record extension has to have a unique name, whether the component is (visibly) inherited from the parent type or declared in the record_extension_part (see 8.3). 9

Examples

Examples of record extensions (of types defined above in 3.9): 10

11
```
type Painted_Point is new Point with
  record
    Paint : Color := White;
  end record;
    -- Components X and Y are inherited
```

12
```
Origin : constant Painted_Point := (X | Y => 0.0, Paint => Black);
```

13
```
type Literal is new Expression with
  record                 -- a leaf in an Expression tree
    Value : Real;
  end record;
```

14
```
type Expr_Ptr is access all Expression'Class;
                                -- see 3.10
```

15
```
type Binary_Operation is new Expression with
  record                 -- an internal node in an Expression tree
    Left, Right : Expr_Ptr;
  end record;
```

16
```
type Addition is new Binary_Operation with null record;
type Subtraction is new Binary_Operation with null record;
  -- No additional components needed for these extensions
```

17
```
Tree : Expr_Ptr :=         -- A tree representation of "5.0 + (13.0–7.0)"
   new Addition'(
      Left  => new Literal'(Value => 5.0),
      Right => new Subtraction'(
         Left  => new Literal'(Value => 13.0),
         Right => new Literal'(Value => 7.0)));
```

Extensions to Ada 83

{*extensions to Ada 83*} Type extension is a new concept. 17.a

Extensions to Ada 95

{*AI95-00344-01*} {*extensions to Ada 95*} Type extensions now can be declared in more nested scopes than their parent types. Additional accessibility checks on allocators and return statements prevent objects from outliving their type. 17.b/2

Wording Changes from Ada 95

{*AI95-00345-01*} Added wording to prevent extending synchronized tagged types. 17.c/2

{*AI95-00391-01*} Defined null extension for use elsewhere. 17.d/2

3.9.2 Dispatching Operations of Tagged Types

1/2 {*AI95-00260-02*} {*AI95-00335-01*} {*dispatching operation* [distributed]} {*dispatching call (on a dispatching operation)*} {*nondispatching call (on a dispatching operation)*} {*statically determined tag*} {*dynamically determined tag*} {*polymorphism*} {*run-time polymorphism*} {*controlling tag (for a call on a dispatching operation)*} The primitive subprograms of a tagged type, the subprograms declared by formal_abstract_-subprogram_declarations, and the stream attributes of a specific tagged type that are available (see 13.13.2) at the end of the declaration list where the type is declared are called *dispatching operations*. [A dispatching operation can be called using a statically determined *controlling* tag, in which case the body to be executed is determined at compile time. Alternatively, the controlling tag can be dynamically determined, in which case the call *dispatches* to a body that is determined at run time;] such a call is termed a *dispatching call*. [As explained below, the properties of the operands and the context of a particular call on a dispatching operation determine how the controlling tag is determined, and hence whether or not the call is a dispatching call. Run-time polymorphism is achieved when a dispatching operation is called by a dispatching call.] {*object-oriented programming (OOP): See dispatching operations of tagged types*} {*OOP (object-oriented programming): See dispatching operations of tagged types*} {*message: See dispatching call*} {*method: See dispatching subprogram*} {*virtual function: See dispatching subprogram*}

1.a.1/2 **Reason:** {*AI95-00335-01*} For the stream attributes of a type declared immediately within a package_specification that has a partial view, the declaration list to consider is the visible part of the package. Stream attributes that are not available in the same declaration list are not dispatching as there is no guarantee that descendants of the type have available attributes (there is such a guarantee for visibly available attributes). If we allowed dispatching for any available attribute, then for attributes defined in the private part we could end up executing a non-existent body.

Language Design Principles

1.a The controlling tag determination rules are analogous to the overload resolution rules, except they deal with run-time type identification (tags) rather than compile-time type resolution. As with overload resolution, controlling tag determination may depend on operands or result context.

Static Semantics

2/2 {*AI95-00260-02*} {*AI95-00416-01*} {*call on a dispatching operation*} {*dispatching operation*} A *call on a dispatching operation* is a call whose name or prefix denotes the declaration of a dispatching operation. {*controlling operand*} A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; {*controlling formal parameter*} the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. {*controlling result*} If the call is to a (primitive) function with result type *T*, then the call has a *controlling result* — the context of the call can control the dispatching. Similarly, if the call is to a function with access result type designating *T*, then the call has a *controlling access result,* and the context can similarly control dispatching.

2.a **Ramification:** This definition implies that a call through the dereference of an access-to-subprogram value is never considered a call on a dispatching operation. Note also that if the prefix denotes a renaming_declaration, the place where the renaming occurs determines whether it is primitive; the thing being renamed is irrelevant.

3 A name or expression of a tagged type is either *statically* tagged, *dynamically* tagged, or *tag indeterminate*, according to whether, when used as a controlling operand, the tag that controls dispatching is determined statically by the operand's (specific) type, dynamically by its tag at run time, or from context. A qualified_expression or parenthesized expression is statically, dynamically, or indeterminately tagged according to its operand. For other kinds of names and expressions, this is determined as follows:

- {*AI95-00416-01*} {*statically tagged*} The name or expression is *statically tagged* if it is of a specific tagged type and, if it is a call with a controlling result or controlling access result, it has at least one statically tagged controlling operand; 4/2

  Discussion: It is illegal to have both statically tagged and dynamically tagged controlling operands in the same call -- see below. 4.a

- {*AI95-00416-01*} {*dynamically tagged*} The name or expression is *dynamically tagged* if it is of a class-wide type, or it is a call with a controlling result or controlling access result and at least one dynamically tagged controlling operand; 5/2

- {*AI95-00416-01*} {*tag indeterminate*} The name or expression is *tag indeterminate* if it is a call with a controlling result or controlling access result, all of whose controlling operands (if any) are tag indeterminate. 6/2

{*8652/0010*} {*AI95-00127-01*} [A type_conversion is statically or dynamically tagged according to whether the type determined by the subtype_mark is specific or class-wide, respectively.] For an object that is designated by an expression whose expected type is an anonymous access-to-specific tagged type, the object is dynamically tagged if the expression, ignoring enclosing parentheses, is of the form X'Access, where X is of a class-wide type, or is of the form **new** T'(...), where T denotes a class-wide subtype. Otherwise, the object is statically or dynamically tagged according to whether the designated type of the type of the expression is specific or class-wide, respectively. 7/1

Ramification: A type_conversion is never tag indeterminate, even if its operand is. A designated object is never tag indeterminate. 7.a

{*8652/0010*} {*AI95-00127-01*} Allocators and access attributes of class-wide types can be used as the controlling parameters of dispatching calls. 7.a.1/1

Legality Rules

A call on a dispatching operation shall not have both dynamically tagged and statically tagged controlling operands. 8

Reason: This restriction is intended to minimize confusion between whether the dynamically tagged operands are implicitly converted to, or tag checked against the specific type of the statically tagged operand(s). 8.a

{*8652/0010*} {*AI95-00127-01*} If the expected type for an expression or name is some specific tagged type, then the expression or name shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. Similarly, if the expected type for an expression is an anonymous access-to-specific tagged type, then the object designated by the expression shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. 9/1

Reason: This prevents implicit "truncation" of a dynamically-tagged value to the specific type of the target object/formal. An explicit conversion is required to request this truncation. 9.a

Ramification: {*AI95-00252-01*} This rule applies to all expressions or names with a specific expected type, not just those that are actual parameters to a dispatching call. This rule does not apply to a membership test whose expression is class-wide, since any type that covers the tested type is explicitly allowed. See 4.5.2. This rule also doesn't apply to a selected_component whose selector_name is a subprogram, since the rules explicitly say that the prefix may be class-wide (see 4.1.3). 9.b/2

{*8652/0011*} {*AI95-00117-01*} {*AI95-00430-01*} In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. {*statically matching (required)* [partial]} If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. {*subtype conformance (required)*} The convention of an inherited dispatching operation is the convention of the corresponding primitive operation of the parent or progenitor type. The default convention of a dispatching operation that overrides an inherited primitive operation is the convention of the inherited operation; if the operation overrides multiple inherited operations, then they 10/2 shall all have the same convention. An explicitly declared dispatching operation shall not be of convention Intrinsic.

10.a **Reason:** These rules ensure that constraint checks can be performed by the caller in a dispatching call, and parameter passing conventions match up properly. A special rule on aggregates prevents values of a tagged type from being created that are outside of its first subtype.

11/2 {*AI95-00416-01*} The default_expression for a controlling formal parameter of a dispatching operation shall be tag indeterminate.

11.a/2 **Reason:** {*AI95-00416-01*} This rule ensures that the default_expression always produces the "correct" tag when called with or without dispatching, or when inherited by a descendant. If it were statically tagged, the default would be useless for a dispatching call; if it were dynamically tagged, the default would be useless for a nondispatching call.

11.1/2 {*AI95-00404-01*} If a dispatching operation is defined by a subprogram_renaming_declaration or the instantiation of a generic subprogram, any access parameter of the renamed subprogram or the generic subprogram that corresponds to a controlling access parameter of the dispatching operation, shall have a subtype that excludes null.

12 A given subprogram shall not be a dispatching operation of two or more distinct tagged types.

12.a **Reason:** This restriction minimizes confusion since multiple dispatching is not provided. The normal solution is to replace all but one of the tagged types with their class-wide types.

12.a.1/1 **Ramification:** {*8652/0098*} {*AI95-00183-01*} This restriction applies even if the partial view (see 7.3) of one or both of the types is untagged. This follows from the definition of dispatching operation: the operation is a dispatching operation anywhere the full views of the (tagged) types are visible.

13 The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see 13.14). [For example, new dispatching operations cannot be added after objects or values of the type exist, nor after deriving a record extension from it, nor after a body.]

13.a/2 **Reason:** {*AI95-00344-01*} This rule is needed because (1) we don't want people dispatching to things that haven't been declared yet, and (2) we want to allow the static part of tagged type descriptors to be static (allocated statically, and initialized to link-time-known symbols). Suppose T2 inherits primitive P from T1, and then overrides P. Suppose P is called *before* the declaration of the overriding P. What should it dispatch to? If the answer is the new P, we've violated the first principle above. If the answer is the old P, we've violated the second principle. (A call to the new one necessarily raises Program_Error, but that's beside the point.)

13.b Note that a call upon a dispatching operation of type *T* will freeze *T*.

13.c We considered applying this rule to all derived types, for uniformity. However, that would be upward incompatible, so we rejected the idea. As in Ada 83, for an untagged type, the above call upon P will call the old P (which is arguably confusing).

13.d/2 **Implementation Note:** {*AI95-00326-01*} Because of this rule, the type descriptor can be created (presumably containing linker symbols pointing at the not-yet-compiled bodies) at the first freezing point of the type. It also prevents, for a (non-incomplete) tagged type declared in a package_specification, overriding in the body or by a child subprogram.

13.e/2 **Ramification:** {*AI95-00251-01*} A consequence is that for a tagged type declaration in a declarative_part, only the last (overriding) primitive subprogram can be declared by a subprogram_body. (Other overridings must be provided by subprogram_declarations.)

Dynamic Semantics

14 {*execution (call on a dispatching operation)* [partial]} {*controlling tag value*} For the execution of a call on a dispatching operation of a type *T*, the *controlling tag value* determines which subprogram body is executed. The controlling tag value is defined as follows:

15 • {*statically determined tag* [partial]} If one or more controlling operands are statically tagged, then the controlling tag value is *statically determined* to be the tag of *T*.

16 • If one or more controlling operands are dynamically tagged, then the controlling tag value is not statically determined, but is rather determined by the tags of the controlling operands. {*Tag_Check* [partial]} {*check, language-defined (Tag_Check)*} If there is more than one dynamically tagged

controlling operand, a check is made that they all have the same tag. {*Constraint_Error (raised by failure of run-time check)*} If this check fails, Constraint_Error is raised unless the call is a function_call whose name denotes the declaration of an equality operator (predefined or user defined) that returns Boolean, in which case the result of the call is defined to indicate inequality, and no subprogram_body is executed. This check is performed prior to evaluating any tag-indeterminate controlling operands.

Reason: Tag mismatch is considered an error (except for "=" and "/=") since the corresponding primitive subprograms in each specific type expect all controlling operands to be of the same type. For tag mismatch with an equality operator, rather than raising an exception, "=" returns False and "/=" returns True. No equality operator is actually invoked, since there is no common tag value to control the dispatch. Equality is a special case to be consistent with the existing Ada 83 principle that equality comparisons, even between objects with different constraints, never raise Constraint_Error. 16.a

- {*AI95-00196-01*} If all of the controlling operands (if any) are tag-indeterminate, then: 17/2
  - {*AI95-00239-01*} {*AI95-00416-01*} If the call has a controlling result or controlling access result and is itself, or designates, a (possibly parenthesized or qualified) controlling operand of an enclosing call on a dispatching operation of a descendant of type *T*, then its controlling tag value is determined by the controlling tag value of this enclosing call; 18/2

Discussion: {*AI95-00239-01*} For code that a user can write explicitly, the only contexts that can control dispatching of a function with a controlling result of type T are those that involve controlling operands of the same type T: if the two types differ there is an illegality and the dynamic semantics are irrelevant. 18.a/2

In the case of an inherited subprogram however, if a default expression is a function call, it may be of type T while the parameter is of a type derived from T. To cover this case, we talk about "a descendant of T" above. This is safe, because if the type of the parameter is descended from the type of the function result, it is guaranteed to inherit or override the function, and this ensures that there will be an appropriate body to dispatch to. Note that abstract functions are not an issue here because the call to the function is a dispatching call, so it is guaranteed to always land on a concrete body. 18.b/2

  - {*AI95-00196-01*} {*AI95-00416-01*} If the call has a controlling result or controlling access result and (possibly parenthesized, qualified, or dereferenced) is the expression of an assignment_statement whose target is of a class-wide type, then its controlling tag value is determined by the target; 18.1/2
  - {*statically determined tag* [partial]} Otherwise, the controlling tag value is statically determined to be the tag of type *T*. 19

Ramification: This includes the cases of a tag-indeterminate procedure call, and a tag-indeterminate function_call that is used to initialize a class-wide formal parameter or class-wide object. 19.a

{*AI95-00345-01*} For the execution of a call on a dispatching operation, the action performed is determined by the properties of the corresponding dispatching operation of the specific type identified by the controlling tag value. If the corresponding operation is explicitly declared for this type, [even if the declaration occurs in a private part], then the action comprises an invocation of the explicit body for the operation. If the corresponding operation is implicitly declared for this type: 20/2

- {*AI95-00345-01*} if the operation is implemented by an entry or protected subprogram (see 9.1 and 9.4), then the action comprises a call on this entry or protected subprogram, with the target object being given by the first actual parameter of the call, and the actual parameters of the entry or protected subprogram being given by the remaining actual parameters of the call, if any; 20.1/2
- {*AI95-00345-01*} otherwise, the action is the same as the action for the corresponding operation of the parent type. 20.2/2

To be honest: In the unusual case in which a dispatching subprogram is explicitly declared (overridden) by a body (with no preceding subprogram_declaration), the body for that dispatching subprogram is that body; that is, the “corresponding explicit body” in the above rule is the body itself. 20.a

Reason: The wording of the above rule is intended to ensure that the same body is executed for a given tag, whether that tag is determined statically or dynamically. For a type declared in a package, it doesn't matter whether a given subprogram is overridden in the visible part or the private part, and it doesn't matter whether the call is inside or outside the package. For example: 20.b

20.c
```
package P1 is
    type T1 is tagged null record;
    procedure Op_A(Arg : in T1);
    procedure Op_B(Arg : in T1);
end P1;
```

20.d
```
with P1; use P1;
package P2 is
    type T2 is new T1 with null record;
    procedure Op_A(Param : in T2);
private
    procedure Op_B(Param : in T2);
end P2;
```

20.e/1
```
with P1; with P2;
procedure Main is
    X : P2.T2;
    Y : P1.T1'Class := X;
begin
    P2.Op_A(Param => X); -- Nondispatching call to a dispatching operation.
    P1.Op_A(Arg => Y); -- Dispatching call.
    P2.Op_B(Arg => X); -- Nondispatching call to a dispatching operation.
    P1.Op_B(Arg => Y); -- Dispatching call.
end Main;
```

20.f The two calls to Op_A both execute the body of Op_A that has to occur in the body of package P2. Similarly, the two calls to Op_B both execute the body of Op_B that has to occur in the body of package P2, even though Op_B is overridden in the private part of P2. Note, however, that the formal parameter names are different for P2.Op_A versus P2.Op_B. The overriding declaration for P2.Op_B is not visible in Main, so the name in the call actually denotes the implicit declaration of Op_B inherited from T1.

20.g If a call occurs in the program text before an overriding, which can happen only if the call is part of a default expression, the overriding will still take effect for that call.

20.h **Implementation Note:** Even when a tag is not *statically determined*, a compiler might still be able to figure it out and thereby avoid the overhead of run-time dispatching.

NOTES

21 72 The body to be executed for a call on a dispatching operation is determined by the tag; it does not matter whether that tag is determined statically or dynamically, and it does not matter whether the subprogram's declaration is visible at the place of the call.

22/2 73 {*AI95-00260-02*} This subclause covers calls on dispatching subprograms of a tagged type. Rules for tagged type membership tests are described in 4.5.2. Controlling tag determination for an assignment_statement is described in 5.2.

23 74 A dispatching call can dispatch to a body whose declaration is not visible at the place of the call.

24 75 A call through an access-to-subprogram value is never a dispatching call, even if the access value designates a dispatching operation. Similarly a call whose prefix denotes a subprogram_renaming_declaration cannot be a dispatching call unless the renaming itself is the declaration of a primitive subprogram.

Extensions to Ada 83

24.a {*extensions to Ada 83*} The concept of dispatching operations is new.

Incompatibilities With Ada 95

24.b/2 {*AI95-00404-01*} {*incompatibilities with Ada 95*} If a dispatching operation is defined by a subprogram_renaming_declaration, and it has a controlling access parameter, Ada 2005 requires the subtype of the parameter to exclude null. The same applies to instantiations. This is required so that all calls to the subprogram operate the same way (controlling access parameters have to exclude null so that dispatching calls will work). Since Ada 95 didn't have the notion of access subtypes that exclude null, and all access parameters excluded null, it had no such rules. These rules will require the addition of an explicit **not null** on nondispatching operations that are later renamed to be dispatching, or on a generic that is used to define a dispatching operation.

Extensions to Ada 95

24.c/2 {*AI95-00416-01*} {*extensions to Ada 95*} Functions that have an access result type can be dispatching in the same way as a function that returns a tagged object directly.

Wording Changes from Ada 95

{*8652/0010*} {*AI95-00127-01*} **Corrigendum:** Allocators and access attributes of objects of class-wide types can be used as the controlling parameter in a dispatching calls. This was an oversight in the definition of Ada 95. (See 3.10.2 and 4.8). 24.d/2

{*8652/0011*} {*AI95-00117-01*} {*AI95-00430-01*} **Corrigendum:** Corrected the conventions of dispatching operations. This is extended in Ada 2005 to cover operations inherited from progenitors, and to ensure that the conventions of all inherited operations are the same. 24.e/2

{*AI95-00196-01*} Clarified the wording to ensure that functions with no controlling operands are tag-indeterminate, and to describe that the controlling tag can come from the target of an assignment_statement. 24.f/2

{*AI95-00239-01*} Fixed the wording to cover default expressions inherited by derived subprograms. A literal reading of the old wording would have implied that operations would be called with objects of the wrong type. 24.g/2

{*AI95-00260-02*} An abstract formal subprogram is a dispatching operation, even though it is not a primitive operation. See 12.6, "Formal Subprograms". 24.h/2

{*AI95-00345-01*} Dispatching calls include operations implemented by entries and protected operations, so we have to update the wording to reflect that. 24.i/2

{*AI95-00335-01*} A stream attribute of a tagged type is usually a dispatching operation, even though it is not a primitive operation. If they weren't dispatching, T'Class'Input and T'Class'Output wouldn't work. 24.j/2

3.9.3 Abstract Types and Subprograms

{*AI95-00345-01*} [{*abstract type*} {*abstract data type (ADT): See also abstract type*} {*ADT (abstract data type): See also abstract type*} {*concrete type: See nonabstract type*} An *abstract type* is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own. {*abstract subprogram*} {*concrete subprogram: See nonabstract subprogram*} An *abstract subprogram* is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body.] 1/2

Glossary entry: {*Abstract type*} An abstract type is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own. 1.a.1/2

Language Design Principles

An abstract subprogram has no body, so the rules in this clause are designed to ensure (at compile time) that the body will never be invoked. We do so primarily by disallowing the creation of values of the abstract type. Therefore, since type conversion and parameter passing don't change the tag, we know we will never get a class-wide value with a tag identifying an abstract type. This means that we only have to disallow nondispatching calls on abstract subprograms (dispatching calls will never reach them). 1.a

Syntax

{*AI95-00218-03*} {*AI95-00348-01*} abstract_subprogram_declaration ::= 1.1/2
  [overriding_indicator]
  subprogram_specification **is abstract**;

Static Semantics

{*AI95-00345-01*} {*abstract type*} {*type (abstract)*} Interface types (see 3.9.4) are abstract types. In addition, a tagged type that has the reserved word **abstract** in its declaration is an abstract type. The class-wide type (see 3.4.1) rooted at an abstract type is not itself an abstract type. 1.2/2

Legality Rules

{*AI95-00345-01*} Only a tagged type shall have the reserved word **abstract** in its declaration. 2/2

Ramification: Untagged types are never abstract, even though they can have primitive abstract subprograms. Such subprograms cannot be called, unless they also happen to be dispatching operations of some tagged type, and then only via a dispatching call. 2.a

2.b Class-wide types are never abstract. If T is abstract, then it is illegal to declare a stand-alone object of type T, but it is OK to declare a stand-alone object of type T'Class; the latter will get a tag from its initial value, and this tag will necessarily be different from T'Tag.

3/2 {*AI95-00260-02*} {*AI95-00348-01*} {*abstract subprogram*} {*subprogram (abstract)*} A subprogram declared by an abstract_subprogram_declaration or a formal_abstract_subprogram_declaration (see 12.6) is an *abstract subprogram*. If it is a primitive subprogram of a tagged type, then the tagged type shall be abstract.

3.a **Ramification:** Note that for a private type, this applies to both views. The following is illegal:

3.b
```
package P is
    type T is abstract tagged private;
    function Foo (X : T) return Boolean is abstract; -- Illegal!
private
    type T is tagged null record; -- Illegal!
    X : T;
    Y : Boolean := Foo (T'Class (X));
end P;
```

3.c The full view of T is not abstract, but has an abstract operation Foo, which is illegal. The two lines marked "-- *Illegal!*" are illegal when taken together.

3.d/2 **Reason:** {*AI95-00310-01*} We considered disallowing untagged types from having abstract primitive subprograms. However, we rejected that plan, because it introduced some silly anomalies, and because such subprograms are harmless. For example:

3.e/1
```
package P is
   type Field_Size is range 0..100;
   type T is abstract tagged null record;
   procedure Print(X : in T; F : in Field_Size := 0) is abstract;
  . . .
package Q is
   type My_Field_Size is new Field_Size;
   -- implicit declaration of Print(X : T; F : My_Field_Size := 0) is abstract;
end Q;
```

3.f It seemed silly to make the derivative of My_Field_Size illegal, just because there was an implicitly declared abstract subprogram that was not primitive on some tagged type. Other rules could be formulated to solve this problem, but the current ones seem like the simplest.

3.g/2 {*AI95-00310-01*} In Ada 2005, abstract primitive subprograms of an untagged type may be used to “undefine” an operation.

3.h/2 **Ramification:** {*AI95-00260-02*} Note that the second sentence does not apply to abstract formal subprograms, as they are never primitive operations of a type.

4/2 {*AI95-00251-01*} {*AI95-00334-01*} {*AI95-00391-01*} If a type has an implicitly declared primitive subprogram that is inherited or is the predefined equality operator, and the corresponding primitive subprogram of the parent or ancestor type is abstract or is a function with a controlling access result, or if a type other than a null extension inherits a function with a controlling result, then:

5/2 • {*AI95-00251-01*} {*AI95-00334-01*} If the type is abstract or untagged, the implicitly declared subprogram is *abstract*.

5.a **Ramification:** Note that it is possible to override a concrete subprogram with an abstract one.

6/2 • {*AI95-00391-01*} Otherwise, the subprogram shall be overridden with a nonabstract subprogram or, in the case of a private extension inheriting a function with a controlling result, have a full type that is a null extension[; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part]. Such a subprogram is said to *require overriding*.{*require overriding*} However, if the type is a generic formal type, the subprogram need not be overridden for the formal type itself; [a nonabstract version will necessarily be provided by the actual type.]

6.a/2 **Reason:** {*AI95-00228-01*} {*AI95-00391-01*} A function that returns the parent type requires overriding for a type extension (or becomes abstract for an abstract type) because conversion from a parent type to a type extension is not defined, and function return semantics is defined in terms of conversion (other than for a null extension; see below). (Note that parameters of mode **in out** or **out** do not have this problem, because the tag of the actual is not changed.)

Note that the overriding required above can be in the private part, which allows the following: 6.b

6.c
```
package Pack1 is
    type Ancestor is abstract ...;
    procedure Do_Something(X : in Ancestor) is abstract;
end Pack1;
```

6.d
```
with Pack1; use Pack1;
package Pack2 is
    type T1 is new Ancestor with record ...;
        -- A concrete type.
    procedure Do_Something(X : in T1); -- Have to override.
end Pack2;
```

6.e
```
with Pack1; use Pack1;
with Pack2; use Pack2;
package Pack3 is
    type T2 is new Ancestor with private;
        -- A concrete type.
private
    type T2 is new T1 with -- Parent different from ancestor.
      record ... end record;
    -- Here, we inherit Pack2.Do_Something.
end Pack3;
```

{*AI95-00228-01*} T2 inherits an abstract Do_Something, but T2 is not abstract, so Do_Something has to be overridden. However, it is OK to override it in the private part. In this case, we override it by inheriting a concrete version from a different type. Nondispatching calls to Pack3.Do_Something are allowed both inside and outside package Pack3, as the client “knows” that the subprogram was necessarily overridden somewhere. 6.f/2

{*AI95-00391-01*} For a null extension, the result of a function with a controlling result is defined in terms of an extension_aggregate with a **null record** extension part (see 3.4). This means that these restrictions on functions with a controlling result do not have to apply to null extensions. 6.g/2

{*AI95-00391-01*} However, functions with controlling access results still require overriding. Changing the tag in place might clobber a preexisting object, and allocating new memory would possibly change the pool of the object, leading to storage leaks. Moreover, copying the object isn't possible for limited types. We don't need to restrict functions that have an access return type of an untagged type, as derived types with primitive subprograms have to have the same representation as their parent type. 6.h/2

A call on an abstract subprogram shall be a dispatching call; [nondispatching calls to an abstract subprogram are not allowed.] 7

Ramification: {*AI95-00310-01*} If an abstract subprogram is not a dispatching operation of some tagged type, then it cannot be called at all. In Ada 2005, such subprograms are not even considered by name resolution (see 6.4). 7.a/2

The type of an aggregate, or of an object created by an object_declaration or an allocator, or a generic formal object of mode **in**, shall not be abstract. The type of the target of an assignment operation (see 5.2) shall not be abstract. The type of a component shall not be abstract. If the result type of a function is abstract, then the function shall be abstract. 8

Reason: This ensures that values of an abstract type cannot be created, which ensures that a dispatching call to an abstract subprogram will not try to execute the nonexistent body. 8.a

Generic formal objects of mode **in** are like constants; therefore they should be forbidden for abstract types. Generic formal objects of mode **in out** are like renamings; therefore, abstract types are OK for them, though probably not terribly useful. 8.b

If a partial view is not abstract, the corresponding full view shall not be abstract. If a generic formal type is abstract, then for each primitive subprogram of the formal that is not abstract, the corresponding primitive subprogram of the actual shall not be abstract. 9

Discussion: By contrast, we allow the actual type to be nonabstract even if the formal type is declared abstract. Hence, the most general formal tagged type possible is "**type** T(<>) **is abstract tagged limited private**;". 9.a

For an abstract private extension declared in the visible part of a package, it is only possible for the full type to be nonabstract if the private extension has no abstract dispatching operations. 9.b

To be honest: {*AI95-00294-01*} In the sentence about primitive subprograms above, there is some ambiguity as to what is meant by “corresponding” in the case where an inherited operation is overridden. This is best explained by an example, where the implicit declarations are shown as comments: 9.c/2

9.d/2
```
package P1 is
   type T1 is abstract tagged null record;
   procedure P (X : T1); -- (1)
end P1;
```

9.e/2
```
package P2 is
   type T2 is abstract new P1.T1 with null record;
   -- procedure P (X : T2); -- (2)
   procedure P (X : T2) is abstract; -- (3)
end P2;
```

9.f/2
```
generic
   type D is abstract new P1.T1 with private;
   -- procedure P (X : D); -- (4)
procedure G (X : D);
```

9.g/2
```
procedure I is new G (P2.T2); -- Illegal.
```

9.h/2 Type T2 inherits a non-abstract procedure P (2) from the primitive procedure P (1) of T1. P (2) is overridden by the explicitly declared abstract procedure P (3). Type D inherits a non-abstract procedure P (4) from P (1). In instantiation I, the operation corresponding to P (4) is the one which is not overridden, that is, P (3): the overridden operation P (2) does not "reemerge". Therefore, the instantiation is illegal.

10 For an abstract type declared in a visible part, an abstract primitive subprogram shall not be declared in the private part, unless it is overriding an abstract subprogram implicitly declared in the visible part. For a tagged type declared in a visible part, a primitive function with a controlling result shall not be declared in the private part, unless it is overriding a function implicitly declared in the visible part.

10.a **Reason:** The "visible part" could be that of a package or a generic package. This rule is needed because a non-abstract type extension declared outside the package would not know about any abstract primitive subprograms or primitive functions with controlling results declared in the private part, and wouldn't know that they need to be overridden with non-abstract subprograms. The rule applies to a tagged record type or record extension declared in a visible part, just as to a tagged private type or private extension. The rule applies to explicitly and implicitly declared abstract subprograms:

10.b
```
package Pack is
    type T is abstract new T1 with private;
private
    type T is abstract new T2 with record ... end record;
    ...
end Pack;
```

10.c The above example would be illegal if T1 has a non-abstract primitive procedure P, but T2 overrides P with an abstract one; the private part should override P with a non-abstract version. On the other hand, if the P were abstract for both T1 and T2, the example would be legal as is.

11/2 {*AI95-00260-02*} A generic actual subprogram shall not be an abstract subprogram unless the generic formal subprogram is declared by a formal_abstract_subprogram_declaration. The prefix of an attribute_reference for the Access, Unchecked_Access, or Address attributes shall not denote an abstract subprogram.

11.a **Ramification:** An abstract_subprogram_declaration is not syntactically a subprogram_declaration. Nonetheless, an abstract subprogram is a subprogram, and an abstract_subprogram_declaration is a declaration of a subprogram.

11.b/2 {*AI95-00260-02*} The part about generic actual subprograms includes those given by default. Of course, an abstract formal subprogram's actual subprogram can be abstract.

Dynamic Semantics

11.1/2 {*AI95-00348-01*} {*elaboration (abstract_subprogram_declaration)* [partial]} The elaboration of an abstract_subprogram_declaration has no effect.

NOTES

12 76 Abstractness is not inherited; to declare an abstract type, the reserved word **abstract** has to be used in the declaration of the type extension.

12.a **Ramification:** A derived type can be abstract even if its parent is not. Similarly, an inherited concrete subprogram can be overridden with an abstract subprogram.

13 77 A class-wide type is never abstract. Even if a class is rooted at an abstract type, the class-wide type for the class is not abstract, and an object of the class-wide type can be created; the tag of such an object will identify some nonabstract type in the class.

Examples

Example of an abstract type representing a set of natural numbers: 14

15
```
package Sets is
    subtype Element_Type is Natural;
    type Set is abstract tagged null record;
    function Empty return Set is abstract;
    function Union(Left, Right : Set) return Set is abstract;
    function Intersection(Left, Right : Set) return Set is abstract;
    function Unit_Set(Element : Element_Type) return Set is abstract;
    procedure Take(Element : out Element_Type;
                   From : in out Set) is abstract;
end Sets;
```

NOTES

16 78 *Notes on the example:* Given the above abstract type, one could then derive various (nonabstract) extensions of the type, representing alternative implementations of a set. One might use a bit vector, but impose an upper bound on the largest element representable, while another might use a hash table, trading off space for flexibility.

16.a **Discussion:** One way to export a type from a package with some components visible and some components private is as follows:

16.b
```
package P is
    type Public_Part is abstract tagged
        record
            ...
        end record;
    type T is new Public_Part with private;
    ...
private
    type T is new Public_Part with
        record
            ...
        end record;
end P;
```

16.c The fact that Public_Part is abstract tells clients they have to create objects of type T instead of Public_Part. Note that the public part has to come first; it would be illegal to declare a private type Private_Part, and then a record extension T of it, unless T were in the private part after the full declaration of Private_Part, but then clients of the package would not have visibility to T.

Extensions to Ada 95

16.d/2 {*AI95-00391-01*} {*extensions to Ada 95*} It is not necessary to override functions with a controlling result for a null extension. This makes it easier to derive a tagged type to complete a private type.

Wording Changes from Ada 95

16.e/2 {*AI95-00251-01*} {*AI95-00345-01*} Updated the wording to reflect the addition of interface types (see 3.9.4).

16.f/2 {*AI95-00260-02*} Updated the wording to reflect the addition of abstract formal subprograms (see 12.6).

16.g/2 {*AI95-00334-01*} The wording of shall-be-overridden was clarified so that it clearly applies to abstract predefined equality.

16.h/2 {*AI95-00348-01*} Moved the syntax and elaboration rule for abstract_subprogram_declaration here, so the syntax and most of the semantics are together (which is consistent with null procedures).

16.i/2 {*AI95-00391-01*} We define the term *require overriding* to make other wording easier to understand.

3.9.4 Interface Types

1/2 {*AI95-00251-01*} {*AI95-00345-01*} [An interface type is an abstract tagged type that provides a restricted form of multiple inheritance. A tagged type, task type, or protected type may have one or more interface types as ancestors.]

1.a/2 **Glossary entry:** {*Interface type*} An interface type is a form of abstract tagged type which has no components or concrete operations except possibly null procedures. Interface types are used for composing other interfaces and tagged types and thereby provide multiple inheritance. Only an interface type can be used as a progenitor of another type.

Language Design Principles

1.b/2 {*AI95-00251-01*} {*AI95-00345-01*} The rules are designed so that an interface can be used as either a parent type or a progenitor type without changing the meaning. That's important so that the order that interfaces are specified in a derived_type_definition is not significant. In particular, we want:

1.c/2
```
    type Con1 is new Int1 and Int2 with null record;
    type Con2 is new Int2 and Int1 with null record;
```

1.d/2 {*AI95-00251-01*} {*AI95-00345-01*} to mean exactly the same thing.

Syntax

2/2 {*AI95-00251-01*} {*AI95-00345-01*} interface_type_definition ::=
[**limited** | **task** | **protected** | **synchronized**] **interface** [**and** interface_list]

3/2 {*AI95-00251-01*} {*AI95-00419-01*} interface_list ::= *interface*_subtype_mark {**and** *interface*_subtype_mark}

Static Semantics

4/2 {*AI95-00251-01*} An interface type (also called an *interface*) is{*interface* [distributed]} {*interface (type)* [partial]} a specific abstract tagged type that is defined by an interface_type_definition.

5/2 {*AI95-00345-01*} An interface with the reserved word **limited**, **task**, **protected**, or **synchronized** in its definition is termed, respectively, a *limited interface*, a *task interface*, a *protected interface*, or a *synchronized interface*. In addition,{*interface (synchronized)* [partial]} {*interface (protected)* [partial]} {*interface (task)* [partial]} {*interface (limited)* [partial]} {*interface (nonlimited)* [partial]} {*synchronized interface*} {*protected interface*} {*task interface*} {*limited interface*} {*nonlimited interface*} all task and protected interfaces are synchronized interfaces, and all synchronized interfaces are limited interfaces.

5.a/2 **Glossary entry:** {*Synchronized*} A synchronized entity is one that will work safely with multiple tasks at one time. A synchronized interface can be an ancestor of a task or a protected type. Such a task or protected type is called a synchronized tagged type.

6/2 {*AI95-00345-01*} {*AI95-00443-01*} {*synchronized tagged type*} {*type (synchronized tagged)* [partial]} {*tagged type (synchronized)* [partial]} {*tagged type (task)* [partial]} {*tagged type (protected)* [partial]} {*task tagged type*} {*protected tagged type*} [A task or protected type derived from an interface is a tagged type.] Such a tagged type is called a *synchronized* tagged type, as are synchronized interfaces and private extensions whose declaration includes the reserved word **synchronized**.

6.a/2 **Proof:** The full definition of tagged types given in 3.9 includes task and protected types derived from interfaces.

6.b/2 **Ramification:** The class-wide type associated with a tagged task type (including a task interface type) is a task type, because "task" is one of the language-defined classes of types (see 3.2. However, the class-wide type associated with an interface is *not* an interface type, as "interface" is *not* one of the language-defined classes (as it is not closed under derivation). In this sense, "interface" is similar to "abstract". The class-wide type associated with an interface is a concrete (nonabstract) indefinite tagged composite type.

6.c/2 "Private extension" includes generic formal private extensions, as explained in 12.5.1.

7/2 {*AI95-00345-01*} A task interface is an [abstract] task type. A protected interface is an [abstract] protected type.

7.a/2 **Proof:** The "abstract" follows from the definition of an interface type.

7.b/2 **Reason:** This ensures that task operations (like abort and the Terminated attribute) can be applied to a task interface type and the associated class-wide type. While there are no protected type operations, we apply the same rule to protected interfaces for consistency.

8/2 {*AI95-00251-01*} [An interface type has no components.]

8.a/2 **Proof:** This follows from the syntax and the fact that discriminants are not allowed for interface types.

9/2 {*AI95-00419-01*} {*progenitor subtype*} {*progenitor type*} An *interface*_subtype_mark in an interface_list names a *progenitor subtype*; its type is the *progenitor type*. An interface type inherits user-defined primitive subprograms from each progenitor type in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4).

Glossary entry: {*Progenitor*} A progenitor of a derived type is one of the types given in the definition of the derived type other than the first. A progenitor is always an interface type. Interfaces, tasks, and protected types may also have progenitors. 9.a.1/2

Legality Rules

{*AI95-00251-01*} All user-defined primitive subprograms of an interface type shall be abstract subprograms or null procedures. 10/2

{*AI95-00251-01*} The type of a subtype named in an interface_list shall be an interface type. 11/2

{*AI95-00251-01*} {*AI95-00345-01*} A type derived from a nonlimited interface shall be nonlimited. 12/2

{*AI95-00345-01*} An interface derived from a task interface shall include the reserved word **task** in its definition; any other type derived from a task interface shall be a private extension or a task type declared by a task declaration (see 9.1). 13/2

{*AI95-00345-01*} An interface derived from a protected interface shall include the reserved word **protected** in its definition; any other type derived from a protected interface shall be a private extension or a protected type declared by a protected declaration (see 9.4). 14/2

{*AI95-00345-01*} An interface derived from a synchronized interface shall include one of the reserved words **task**, **protected**, or **synchronized** in its definition; any other type derived from a synchronized interface shall be a private extension, a task type declared by a task declaration, or a protected type declared by a protected declaration. 15/2

Reason: We require that an interface descendant of a task, protected, or synchronized interface repeat the explicit kind of interface it will be, rather than simply inheriting it, so that a reader is always aware of whether the interface provides synchronization and whether it may be implemented only by a task or protected type. The only place where inheritance of the kind of interface might be useful would be in a generic if you didn't know the kind of the actual interface. However, the value of that is low because you cannot implement an interface properly if you don't know whether it is a task, protected, or synchronized interface. Hence, we require the kind of the actual interface to match the kind of the formal interface (see 12.5.5). 15.a/2

{*AI95-00345-01*} No type shall be derived from both a task interface and a protected interface. 16/2

Reason: This prevents a single private extension from inheriting from both a task and a protected interface. For a private type, there can be no legal completion. For a generic formal derived type, there can be no possible matching type (so no instantiation could be legal). This rule provides early detection of the errors. 16.a

{*AI95-00251-01*} In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.{*generic contract issue* [partial]} 17/2

Ramification: This paragraph is intended to apply to all of the Legality Rules in this clause. We cannot allow interface types which do not obey these rules, anywhere. Luckily, deriving from a formal type (which might be an interface) is not allowed for any tagged types in a generic body. So checking in the private part of a generic covers all of the cases. 17.a/2

Dynamic Semantics

{*AI95-00251-01*} The elaboration of an interface_type_definition has no effect. 18/2

Discussion: An interface_list is made up of subtype_marks, which do not need to be elaborated, so the interface_list does not either. This is consistent with the handling of discriminant_parts. 18.a/2

NOTES

79 {*AI95-00411-01*} Nonlimited interface types have predefined nonabstract equality operators. These may be overridden with user-defined abstract equality operators. Such operators will then require an explicit overriding for any nonabstract descendant of the interface. 19/2

Examples

{*AI95-00433-01*} *Example of a limited interface and a synchronized interface extending it:* 20/2

21/2
```
type Queue is limited interface;
procedure Append(Q : in out Queue; Person : in Person_Name) is abstract;
procedure Remove_First(Q      : in out Queue;
                       Person : out Person_Name) is abstract;
function Cur_Count(Q : in Queue) return Natural is abstract;
function Max_Count(Q : in Queue) return Natural is abstract;
-- See 3.10.1 for Person_Name.
```

22/2
```
Queue_Error : exception;
-- Append raises Queue_Error if Count(Q) = Max_Count(Q)
-- Remove_First raises Queue_Error if Count(Q) = 0
```

23/2
```
type Synchronized_Queue is synchronized interface and Queue; -- see 9.11
procedure Append_Wait(Q      : in out Synchronized_Queue;
                      Person : in Person_Name) is abstract;
procedure Remove_First_Wait(Q      : in out Synchronized_Queue;
                            Person : out Person_Name) is abstract;
```

24/2
```
...
```

25/2
```
procedure Transfer(From   : in out Queue'Class;
                   To     : in out Queue'Class;
                   Number : in     Natural := 1) is
   Person : Person_Name;
begin
   for I in 1..Number loop
      Remove_First(From, Person);
      Append(To, Person);
   end loop;
end Transfer;
```

26/2 This defines a Queue interface defining a queue of people. (A similar design could be created to define any kind of queue simply by replacing Person_Name by an appropriate type.) The Queue interface has four dispatching operations, Append, Remove_First, Cur_Count, and Max_Count. The body of a class-wide operation, Transfer is also shown. Every non-abstract extension of Queue must provide implementations for at least its four dispatching operations, as they are abstract. Any object of a type derived from Queue may be passed to Transfer as either the From or the To operand. The two operands need not be of the same type in any given call.

27/2 The Synchronized_Queue interface inherits the four dispatching operations from Queue and adds two additional dispatching operations, which wait if necessary rather than raising the Queue_Error exception. This synchronized interface may only be implemented by a task or protected type, and as such ensures safe concurrent access.

28/2 {*AI95-00433-01*} *Example use of the interface:*

29/2
```
type Fast_Food_Queue is new Queue with record ...;
procedure Append(Q : in out Fast_Food_Queue; Person : in Person_Name);
procedure Remove_First(Q : in out Fast_Food_Queue; Person : in Person_Name);
function Cur_Count(Q : in Fast_Food_Queue) return Natural;
function Max_Count(Q : in Fast_Food_Queue) return Natural;
```

30/2
```
...
```

31/2
```
Cashier, Counter : Fast_Food_Queue;
```

32/2
```
...
-- Add George (see 3.10.1) to the cashier's queue:
Append (Cashier, George);
-- After payment, move George to the sandwich counter queue:
Transfer (Cashier, Counter);
...
```

33/2 An interface such as Queue can be used directly as the parent of a new type (as shown here), or can be used as a progenitor when a type is derived. In either case, the primitive operations of the interface are inherited. For Queue, the implementation of the four inherited routines must be provided. Inside the call of Transfer, calls will dispatch to the implementations of Append and Remove_First for type Fast_Food_Queue.

34/2 {*AI95-00433-01*} *Example of a task interface:*

35/2
```
type Serial_Device is task interface;  -- see 9.1
procedure Read (Dev : in Serial_Device; C : out Character) is abstract;
procedure Write(Dev : in Serial_Device; C : in  Character) is abstract;
```

36/2 The Serial_Device interface has two dispatching operations which are intended to be implemented by task entries (see 9.1).

Extensions to Ada 95

36.a/2 {*AI95-00251-01*} {*AI95-00345-01*} {*extensions to Ada 95*} Interface types are new. They provide multiple inheritance of interfaces, similar to the facility provided in Java and other recent language designs.

3.10 Access Types

1 {*access type*} {*access value*} {*designate*} A value of an access type (an *access value*) provides indirect access to the object or subprogram it *designates*. Depending on its type, an access value can designate either subprograms, objects created by allocators (see 4.8), or more generally *aliased* objects of an appropriate type. {*pointer: See access value*} {*pointer type: See access type*}

1.a **Discussion:** A name *denotes* an entity; an access value *designates* an entity. The “dereference” of an access value X, written “X.**all**”, is a name that denotes the entity designated by X.

Language Design Principles

1.b Access values should always be well defined (barring uses of certain unchecked features of Section 13). In particular, uninitialized access variables should be prevented by compile-time rules.

Syntax

2/2 {*AI95-00231-01*} access_type_definition ::=
  [null_exclusion] access_to_object_definition
 | [null_exclusion] access_to_subprogram_definition

3 access_to_object_definition ::=
  access [general_access_modifier] subtype_indication

4 general_access_modifier ::= **all** | **constant**

5 access_to_subprogram_definition ::=
  access [**protected**] **procedure** parameter_profile
 | **access** [**protected**] **function** parameter_and_result_profile

5.1/2 {*AI95-00231-01*} null_exclusion ::= **not null**

6/2 {*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00404-01*} access_definition ::=
  [null_exclusion] **access** [**constant**] subtype_mark
 | [null_exclusion] **access** [**protected**] **procedure** parameter_profile
 | [null_exclusion] **access** [**protected**] **function** parameter_and_result_profile

Static Semantics

7/1 {*8652/0012*} {*AI95-00062-01*} {*access-to-object type*} {*access-to-subprogram type*} {*pool-specific access type*} {*general access type*} There are two kinds of access types, *access-to-object* types, whose values designate objects, and *access-to-subprogram* types, whose values designate subprograms. {*storage pool*} Associated with an access-to-object type is a *storage pool*; several access types may share the same storage pool. All descendants of an access type share the same storage pool. {*pool element*} A storage pool is an area of storage used to hold dynamically allocated objects (called *pool elements*) created by allocators[; storage pools are described further in 13.11, “Storage Management”].

8 {*pool-specific access type*} {*general access type*} Access-to-object types are further subdivided into *pool-specific* access types, whose values can designate only the elements of their associated storage pool, and *general* access types, whose values can designate the elements of any storage pool, as well as aliased objects created by declarations rather than allocators, and aliased subcomponents of other objects.

8.a **Implementation Note:** The value of an access type will typically be a machine address. However, a value of a pool-specific access type can be represented as an offset (or index) relative to its storage pool, since it can point only to the elements of that pool.

9/2 {*AI95-00225-01*} {*AI95-00363-01*} {*aliased*} A view of an object is defined to be *aliased* if it is defined by an object_declaration or component_definition with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. The current instance of a limited tagged type, a protected type, a task type, or a type that has the reserved word **limited** in its full definition is also defined to be aliased. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. [Aliased views are the ones that can be designated by an access value.]

9.a **Glossary entry:** {*Aliased*} An aliased view of an object is one that can be designated by an access value. Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word **aliased**. The Access attribute can be used to create an access value designating an aliased object.

9.b **Ramification:** The current instance of a nonlimited type is not aliased.

9.c The object created by an allocator is aliased, but not its subcomponents, except of course for those that themselves have **aliased** in their component_definition.

9.d The renaming of an aliased object is aliased.

9.e Slices are never aliased. See 4.1.2 for more discussion.

9.f/2 **Reason:** {*AI95-00225-01*} The current instance of a limited type is defined to be aliased so that an access discriminant of a component can be initialized with T'Access inside the definition of T. Note that we don't want this to apply to a type that could become nonlimited later within its immediate scope, so we require the full definition to be limited.

9.g A formal parameter of a tagged type is defined to be aliased so that a (tagged) parameter X may be passed to an access parameter P by using P => X'Access. Access parameters are most important for tagged types because of dispatching-on-access-parameters (see 3.9.2). By restricting this to formal parameters, we minimize problems associated with allowing components that are not declared aliased to be pointed-to from within the same record.

9.h A view conversion of an aliased view is aliased so that the type of an access parameter can be changed without first converting to a named access type. For example:

9.i
```
type T1 is tagged ...;
procedure P(X : access T1);
```

9.j
```
type T2 is new T1 with ...;
procedure P(X : access T2) is
begin
    P(T1(X.all)'Access);  -- hand off to T1's P
    . . .      -- now do extra T2-specific processing
end P;
```

9.k/2 *This paragraph was deleted.*{*AI95-00363-01*}

9.l/2 We considered making more kinds of objects aliased by default. In particular, any object of a by-reference type will pretty much have to be allocated at an addressable location, so it can be passed by reference without using bit-field pointers. Therefore, one might wish to allow the Access and Unchecked_Access attributes for such objects. However, private parts are transparent to the definition of “by-reference type”, so if we made all objects of a by-reference type aliased, we would be violating the privacy of private parts. Instead, we would have to define a concept of “visibly by-reference” and base the rule on that. This seemed to complicate the rules more than it was worth, especially since there is no way to declare an untagged limited private type to be by-reference, since the full type might by nonlimited.

9.m **Discussion:** Note that we do not use the term “aliased” to refer to formal parameters that are referenced through multiple access paths (see 6.2).

10 An access_to_object_definition defines an access-to-object type and its first subtype; {*designated subtype (of a named access type)*} {*designated type (of a named access type)*} the subtype_indication defines the

designated subtype of the access type. If a general_access_modifier appears, then the access type is a general access type. {*access-to-constant type*} If the modifier is the reserved word **constant**, then the type is an *access-to-constant type*[; a designated object cannot be updated through a value of such a type]. {*access-to-variable type*} If the modifier is the reserved word **all**, then the type is an *access-to-variable type*[; a designated object can be both read and updated through a value of such a type]. If no general_-access_modifier appears in the access_to_object_definition, the access type is a pool-specific access-to-variable type.

To be honest: The type of the designated subtype is called the *designated type*. 10.a

Reason: The modifier **all** was picked to suggest that values of a general access type could point into “all” storage pools, as well as to objects declared aliased, and that “all” access (both read and update) to the designated object was provided. We couldn't think of any use for pool-specific access-to-constant types, so any access type defined with the modifier **constant** is considered a general access type, and can point into any storage pool or at other (appropriate) aliased objects. 10.b

Implementation Note: The predefined generic Unchecked_Deallocation can be instantiated for any named access-to-variable type. There is no (language-defined) support for deallocating objects designated by a value of an access-to-constant type. Because of this, an allocator for an access-to-constant type can allocate out of a storage pool with no support for deallocation. Frequently, the allocation can be done at link-time, if the size and initial value are known then. 10.c

Discussion: For the purpose of generic formal type matching, the relevant subclasses of access types are access-to-subprogram types, access-to-constant types, and (named) access-to-variable types, with its subclass (named) general access-to-variable types. Pool-specific access-to-variable types are not a separately matchable subclass of types, since they don't have any “extra” operations relative to all (named) access-to-variable types. 10.d

{*access-to-subprogram type*} An access_to_subprogram_definition defines an access-to-subprogram type and its first subtype; {*designated profile (of an access-to-subprogram type)*} the parameter_profile or parameter_and_result_profile defines the *designated profile* of the access type. {*calling convention (associated with a designated profile)*} There is a *calling convention* associated with the designated profile[; only subprograms with this calling convention can be designated by values of the access type.] By default, the calling convention is “*protected*” if the reserved word **protected** appears, and “Ada” otherwise. [See Annex B for how to override this default.] 11

Ramification: The calling convention *protected* is in italics to emphasize that it cannot be specified explicitly by the user. This is a consequence of it being a reserved word. 11.a

Implementation Note: {*AI95-00254-01*} For a named access-to-subprogram type, the representation of an access value might include implementation-defined information needed to support up-level references — for example, a static link. The accessibility rules (see 3.10.2) ensure that in a "global-display-based" implementation model (as opposed to a static-link-based model), a named access-to-(unprotected)-subprogram value need consist only of the address of the subprogram. The global display is guaranteed to be properly set up any time the designated subprogram is called. Even in a static-link-based model, the only time a static link is definitely required is for an access-to-subprogram type declared in a scope nested at least two levels deep within subprogram or task bodies, since values of such a type might designate subprograms nested a smaller number of levels. For the normal case of a named access-to-subprogram type declared at the outermost (library) level, a code address by itself should be sufficient to represent the access value in many implementations. 11.b/2

For access-to-protected-subprogram, the access values will necessarily include both an address (or other identification) of the code of the subprogram, as well as the address of the associated protected object. This could be thought of as a static link, but it will be needed even for global-display-based implementation models. It corresponds to the value of the “implicit parameter” that is passed into every call of a protected operation, to identify the current instance of the protected type on which they are to operate. 11.c

Any Elaboration_Check is performed when a call is made through an access value, rather than when the access value is first "created" via a 'Access. For implementation models that normally put that check at the call-site, an access value will have to point to a separate entry point that does the check. Alternatively, the access value could point to a "subprogram descriptor" that consisted of two words (or perhaps more), the first being the address of the code, the second being the elaboration bit. Or perhaps more efficiently, just the address of the code, but using the trick that the descriptor is initialized to point to a Raise-Program-Error routine initially, and then set to point to the "real" code when the body is elaborated. 11.d

For implementations that share code between generic instantiations, the extra level of indirection suggested above to support Elaboration_Checks could also be used to provide a pointer to the per-instance data area normally required 11.e when calling shared code. The trick would be to put a pointer to the per-instance data area into the subprogram descriptor, and then make sure that the address of the subprogram descriptor is loaded into a "known" register whenever an indirect call is performed. Once inside the shared code, the address of the per-instance data area can be retrieved out of the subprogram descriptor, by indexing off the "known" register.

11.f/2 *This paragraph was deleted.*{*AI95-00344-01*}

11.g/2 {*AI95-00254-01*} Note that access parameters of an anonymous access-to-subprogram type are permitted. Such parameters represent full “downward” closures, meaning that in an implementation that uses a per-task (global) display, the display will have to be passed as a hidden parameter, and reconstructed at the point of call.

12/2 {*AI95-00230-01*} {*AI95-00231-01*} {*AI95-00254-01*} {*anonymous access type*} {*designated subtype (of an anonymous access type)*} {*designated type (of an anonymous access type)*} An access_definition defines an anonymous general access type or an anonymous access-to-subprogram type. For a general access type, the subtype_mark denotes its *designated subtype*; if the general_access_modifier **constant** appears, the type is an access-to-constant type; otherwise it is an access-to-variable type. For an access-to-subprogram type, the parameter_profile or parameter_and_result_profile denotes its *designated profile*.{*designated profile (of an anonymous access type)*}

13/2 {*AI95-00230-01*} {*AI95-00231-01*} {*null value (of an access type)*} For each access type, there is a null access value designating no entity at all, which can be obtained by (implicitly) converting the literal **null** to the access type. [The null value of an access type is the default initial value of the type.] Non-null values of an access-to-object type are obtained by evaluating an allocator[, which returns an access value designating a newly created object (see 3.10.2)], or in the case of a general access-to-object type, evaluating an attribute_reference for the Access or Unchecked_Access attribute of an aliased view of an object. Non-null values of an access-to-subprogram type are obtained by evaluating an attribute_reference for the Access attribute of a non-intrinsic subprogram..

13.a/2 *This paragraph was deleted.*{*AI95-00231-01*}

13.b/2 *This paragraph was deleted.*{*AI95-00231-01*}

13.1/2 {*AI95-00231-01*} {*excludes null (subtype)*} A null_exclusion in a construct specifies that the null value does not belong to the access subtype defined by the construct, that is, the access subtype *excludes null*. In addition, the anonymous access subtype defined by the access_definition for a controlling access parameter (see 3.9.2) excludes null. Finally, for a subtype_indication without a null_exclusion, the subtype denoted by the subtype_indication excludes null if and only if the subtype denoted by the subtype_mark in the subtype_indication excludes null.

13.c/2 **Reason:** {*AI95-00231-01*} An access_definition used in a controlling parameter excludes null because it is necessary to read the tag to dispatch, and null has no tag. We would have preferred to require **not null** to be specified for such parameters, but that would have been too incompatible with Ada 95 code to require.

13.d/2 {*AI95-00416-01*} Note that we considered imposing a similar implicit null exclusion for controlling access results, but chose not to do that, because there is no Ada 95 compatibility issue, and there is no automatic null check inherent in the use of a controlling access result. If a null check is necessary, it is because there is a dereference of the result, or because the value is passed to a parameter whose subtype excludes null. If there is no dereference of the result, a null return value is perfectly acceptable, and can be a useful indication of a particular status of the call.

14/1 {*8652/0013*} {*AI95-00012-01*} {*constrained (subtype)* [partial]} {*unconstrained (subtype)* [partial]} [All subtypes of an access-to-subprogram type are constrained.] The first subtype of a type defined by an access_definition or an access_to_object_definition is unconstrained if the designated subtype is an unconstrained array or discriminated subtype; otherwise it is constrained.

14.a **Proof:** The Legality Rules on range_constraints (see 3.5) do not permit the subtype_mark of the subtype_indication to denote an access-to-scalar type, only a scalar type. The Legality Rules on index_constraints (see 3.6.1) and discriminant_constraints (see 3.7.1) both permit access-to-composite types in a subtype_indication with such _constraints. Note that an access-to-access-to-composite is never permitted in a subtype_indication with a constraint.

14.b/2 **Reason:** {*AI95-00363-01*} Only composite_constraints are permitted for an access type, and only on access-to-composite types. A constraint on an access-to-scalar or access-to-access type might be violated due to assignments via other access paths that were not so constrained. By contrast, if the designated subtype is an array or discriminated type without defaults, the constraint could not be violated by unconstrained assignments, since array objects are always constrained, and discriminated objects are also constrained when the type does not have defaults for its discriminants. Constraints are not allowed on general access-to-unconstrained discriminated types if the type has defaults for its discriminants; constraints on pool-specific access types are usually allowed because allocated objects are usually constrained by their initial value.

Legality Rules

{*AI95-00231-01*} If a subtype_indication, discriminant_specification, parameter_specification, parameter_and_result_profile, object_renaming_declaration, or formal_object_declaration has a null_-exclusion, the subtype_mark in that construct shall denote an access subtype that does not exclude null. 14.1/2

To be honest: {*AI95-00231-01*} This means "directly allowed in"; we are not talking about a null_exclusion that occurs in an access_definition in one of these constructs (for an access_definition, the subtype_mark in such an access_definition is not restricted). 14.c/2

Reason: {*AI95-00231-01*} This is similar to doubly constraining a composite subtype, which we also don't allow. 14.d/2

Dynamic Semantics

{*AI95-00231-01*} {*compatibility (composite_constraint with an access subtype)* [partial]} A composite_constraint is *compatible* with an unconstrained access subtype if it is compatible with the designated subtype. A null_exclusion is compatible with any access subtype that does not exclude null. {*satisfies (for an access value)* [partial]} An access value *satisfies* a composite_constraint of an access subtype if it equals the null value of its type or if it designates an object whose value satisfies the constraint. An access value satisfies an exclusion of the null value if it does not equal the null value of its type. 15/2

{*elaboration (access_type_definition)* [partial]} The elaboration of an access_type_definition creates the access type and its first subtype. For an access-to-object type, this elaboration includes the elaboration of the subtype_indication, which creates the designated subtype. 16

{*AI95-00230-01*} {*AI95-00254-01*} {*elaboration (access_definition)* [partial]} The elaboration of an access_definition creates an anonymous access type. 17/2

NOTES

80 Access values are called "pointers" or "references" in some other languages. 18

81 Each access-to-object type has an associated storage pool; several access types can share the same pool. An object can be created in the storage pool of an access type by an allocator (see 4.8) for the access type. A storage pool (roughly) corresponds to what some other languages call a "heap." See 13.11 for a discussion of pools. 19

82 Only index_constraints and discriminant_constraints can be applied to access types (see 3.6.1 and 3.7.1). 20

Examples

Examples of access-to-object types: 21

22/2
```
{AI95-00433-01} type Peripheral_Ref is not null access Peripheral;  -- see 3.8.1
type Binop_Ptr is access all Binary_Operation'Class;
                                          -- general access-to-class-wide, see 3.9.1
```

Example of an access subtype: 23

24
```
subtype Drum_Ref is Peripheral_Ref(Drum);  -- see 3.8.1
```

25 *Example of an access-to-subprogram type:*

26
```
type Message_Procedure is access procedure (M : in String := "Error!");
procedure Default_Message_Procedure(M : in String);
Give_Message : Message_Procedure := Default_Message_Procedure'Access;
...
procedure Other_Procedure(M : in String);
...
Give_Message := Other_Procedure'Access;
...
Give_Message("File not found.");   -- call with parameter (.all is optional)
Give_Message.all;                  -- call with no parameters
```

Extensions to Ada 83

26.a {*extensions to Ada 83*} The syntax for access_type_definition is changed to support general access types (including access-to-constants) and access-to-subprograms. The syntax rules for general_access_modifier and access_definition are new.

Wording Changes from Ada 83

26.b We use the term "storage pool" to talk about the data area from which allocation takes place. The term "collection" is no longer used. ("Collection" and "storage pool" are not the same thing because multiple unrelated access types can share the same storage pool; see 13.11 for more discussion.)

Inconsistencies With Ada 95

26.c/2 {*AI95-00231-01*} {*inconsistencies with Ada 95*} Access discriminants and non-controlling access parameters no longer exclude null. A program which passed **null** to such an access discriminant or access parameter and expected it to raise Constraint_Error may fail when compiled with Ada 2005. One hopes that there no such programs outside of the ACATS. (Of course, a program which actually wants to pass **null** will work, which is far more likely.)

26.d/2 {*AI95-00363-01*} Most unconstrained aliased objects with defaulted discriminants are no longer constrained by their initial values. This means that a program that raised Constraint_Error from an attempt to change the discriminants will no longer do so. The change only affects programs that depended on the raising of Constraint_Error in this case, so the inconsistency is unlikely to occur outside of the ACATS. This change may however cause compilers to implement these objects differently, possibly taking additional memory or time. This is unlikely to be worse than the differences caused by any major compiler upgrade.

Incompatibilities With Ada 95

26.e/2 {*AI95-00225-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** The rule defining when a current instance of a limited type is considered to be aliased has been tightened to apply only to types that cannot become nonlimited. A program that attempts to take 'Access of the current instance of a limited type that can become nonlimited will be illegal in Ada 2005. While original Ada 95 allowed the current instance of any limited type to be treated as aliased, this was inconsistently implemented in compilers, and was likely to not work as expected for types that are ultimately nonlimited.

Extensions to Ada 95

26.f/2 {*AI95-00231-01*} {*extensions to Ada 95*} The null_exclusion is new. It can be used in both anonymous and named access type definitions. It is most useful to declare that parameters cannot be **null**, thus eliminating the need for checks on use.

26.g/2 {*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00404-01*} The kinds of anonymous access types allowed were increased by adding anonymous access-to-constant and anonymous access-to-subprogram types. Anonymous access-to-subprogram types used as parameters allow passing of subprograms at any level.

Wording Changes from Ada 95

26.h/2 {*8652/0012*} {*AI95-00062-01*} **Corrigendum:** Added accidentally-omitted wording that says that a derived access type shares its storage pool with its parent type. This was clearly intended, both because of a note in 3.4, and because anything else would have been incompatible with Ada 83.

26.i/2 {*8652/0013*} {*AI95-00012-01*} **Corrigendum:** Fixed typographical errors in the description of when access types are constrained.

26.j/2 {*AI95-00230-01*} The wording was fixed to allow allocators and the literal **null** for anonymous access types. The former was clearly intended by Ada 95; see the Implementation Advice in 13.11.

{*AI95-00363-01*} The rules about aliased objects being constrained by their initial values now apply only to allocated objects, and thus have been moved to 4.8, "Allocators". 26.k/2

3.10.1 Incomplete Type Declarations

1 There are no particular limitations on the designated type of an access type. In particular, the type of a component of the designated type can be another access type, or even the same access type. This permits mutually dependent and recursive access types. An incomplete_type_declaration can be used to introduce a type to be used as a designated type, while deferring its full definition to a subsequent full_type_declaration.

Syntax

2/2 {*AI95-00326-01*} incomplete_type_declaration ::=
type defining_identifier [discriminant_part] [**is tagged**];

Static Semantics

2.1/2 {*AI95-00326-01*} {*incomplete type*} {*incomplete view*} An incomplete_type_declaration declares an *incomplete view* of a type and its first subtype; the first subtype is unconstrained if a discriminant_part appears. If the incomplete_type_declaration includes the reserved word **tagged**, it declares a *tagged incomplete view*.{*incomplete view (tagged)*} {*tagged incomplete view*} An incomplete view of a type is a limited view of the type (see 7.5).

2.2/2 {*AI95-00326-01*} Given an access type *A* whose designated type *T* is an incomplete view, a dereference of a value of type *A* also has this incomplete view except when:

- 2.3/2 it occurs within the immediate scope of the completion of *T*, or
- 2.4/2 it occurs within the scope of a nonlimited_with_clause that mentions a library package in whose visible part the completion of *T* is declared.

2.5/2 In these cases, the dereference has the full view of *T*.

2.a/2 **Discussion:** We need the "in whose visible part" rule so that the second rule doesn't trigger in the body of a package with a **with** of a child unit:

2.b/2
```
package P is
private
   type T;
   type PtrT is access T;
end P;
```

2.c/2
```
private package P.C is
   Ptr : PtrT;
end P.C;
```

2.d/2
```
with P.C;
package body P is
    -- Ptr.all'Size is not legal here, but it is in the scope of a
    -- nonlimited_with_clause for P.
    type T is ...
    --  Ptr.all'Size is legal here.
end P;
```

2.6/2 {*AI95-00412-01*} Similarly, if a subtype_mark denotes a subtype_declaration defining a subtype of an incomplete view *T*, the subtype_mark denotes an incomplete view except under the same two circumstances given above, in which case it denotes the full view of *T*.

Legality Rules

3 {*requires a completion (incomplete_type_declaration)* [partial]} An incomplete_type_declaration requires a completion, which shall be a full_type_declaration. [If the incomplete_type_declaration occurs immediately within either the visible part of a package_specification or a declarative_part, then the

full_type_declaration shall occur later and immediately within this visible part or declarative_part. If the incomplete_type_declaration occurs immediately within the private part of a given package_-specification, then the full_type_declaration shall occur later and immediately within either the private part itself, or the declarative_part of the corresponding package_body.]

3.a **Proof:** This is implied by the next AARM-only rule, plus the rules in 3.11.1, "Completions of Declarations" which require a completion to appear later and immediately within the same declarative region.

3.b **To be honest:** If the incomplete_type_declaration occurs immediately within the visible part of a package_specification, then the full_type_declaration shall occur immediately within this visible part.

3.c **To be honest:** If the implementation supports it, an incomplete_type_declaration can be completed by a pragma Import.

4/2 {*AI95-00326-01*} If an incomplete_type_declaration includes the reserved word **tagged**, then a full_-type_declaration that completes it shall declare a tagged type. If an incomplete_type_declaration has a known_discriminant_part, then a full_type_declaration that completes it shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1). {*full conformance (required)*} [If an incomplete_type_-declaration has no discriminant_part (or an unknown_discriminant_part), then a corresponding full_-type_declaration is nevertheless allowed to have discriminants, either explicitly, or inherited via derivation.]

5/2 {*AI95-00326-01*} A name that denotes an incomplete view of a type may be used as follows:

6 • as the subtype_mark in the subtype_indication of an access_to_object_definition; [the only form of constraint allowed in this subtype_indication is a discriminant_constraint;]

6.a **Implementation Note:** We now allow discriminant_constraints even if the full type is deferred to the package body. However, there is no particular implementation burden because we have dropped the concept of the dependent compatibility check. In other words, we have effectively repealed AI83-00007.

7/2 • {*AI95-00326-01*} {*AI95-00412-01*} as the subtype_mark in the subtype_indication of a subtype_declaration; the subtype_indication shall not have a null_exclusion or a constraint;

8/2 • {*AI95-00326-01*} as the subtype_mark in an access_definition.

8.1/2 {*AI95-00326-01*} If such a name denotes a tagged incomplete view, it may also be used:

8.2/2 • {*AI95-00326-01*} as the subtype_mark defining the subtype of a parameter in a formal_part;

9/2 • {*AI95-00326-01*} as the prefix of an attribute_reference whose attribute_designator is Class; such an attribute_reference is restricted to the uses allowed here; it denotes a tagged incomplete view.

9.a/2 *This paragraph was deleted.*{*AI95-00326-01*}

9.1/2 {*AI95-00326-01*} If such a name occurs within the declaration list containing the completion of the incomplete view, it may also be used:

9.2/2 • {*AI95-00326-01*} as the subtype_mark defining the subtype of a parameter or result of an access_to_subprogram_definition.

9.a.1/2 **Reason:** This allows, for example, a record to have a component designating a subprogram that takes that same record type as a parameter.

9.3/2 {*AI95-00326-01*} If any of the above uses occurs as part of the declaration of a primitive subprogram of the incomplete view, and the declaration occurs immediately within the private part of a package, then the completion of the incomplete view shall also occur immediately within the private part; it shall not be deferred to the package body.

9.b/2 **Reason:** This fixes a hole in Ada 95 where a dispatching operation with an access parameter could be declared in a private part and a dispatching call on it could occur in a child even though there is no visibility on the full type, requiring access to the controlling tag without access to the representation of the type.

9.4/2 {*AI95-00326-01*} No other uses of a name that denotes an incomplete view of a type are allowed.

{*AI95-00326-01*} A prefix that denotes an object shall not be of an incomplete view. 10/2

Reason: We used to disallow all dereferences of an incomplete type. Now we only disallow such dereferences when used as a prefix. Dereferences used in other contexts do not pose a problem since normal type matching will preclude their use except when the full type is “nearby” as context (for example, as the expected type). 10.a/2

This also disallows prefixes that are directly of an incomplete view. For instance, a parameter *P* can be declared of a tagged incomplete type, but we don't want to allow *P*'Size, *P*'Alignment, or the like, as representation values aren't known for an incomplete view. 10.b/2

We say “denotes an object” so that prefixes that directly name an incomplete view are not covered; the previous rules cover such cases, and we certainly don't want to ban Incomp'Class. 10.c/2

Static Semantics

This paragraph was deleted.{*AI95-00326-01*} 11/2

Dynamic Semantics

{*elaboration (incomplete_type_declaration)* [partial]} The elaboration of an incomplete_type_declaration has no effect. 12

Reason: An incomplete type has no real existence, so it doesn't need to be "created" in the usual sense we do for other types. It is roughly equivalent to a "forward;" declaration in Pascal. Private types are different, because they have a different set of characteristics from their full type. 12.a

NOTES

83 {*completion legality* [partial]} Within a declarative_part, an incomplete_type_declaration and a corresponding full_type_declaration cannot be separated by an intervening body. This is because a type has to be completely defined before it is frozen, and a body freezes all types declared prior to it in the same declarative_part (see 13.14). 13

Examples

Example of a recursive type: 14

15
```
type Cell;  --  incomplete type declaration
type Link is access Cell;
```

16
```
type Cell is
   record
      Value  : Integer;
      Succ   : Link;
      Pred   : Link;
   end record;
```

17
```
Head   : Link  := new Cell'(0, null, null);
Next   : Link  := Head.Succ;
```

Examples of mutually dependent access types: 18

19/2
```
{AI95-00433-01} type Person(<>);     -- incomplete type declaration
type Car is tagged; -- incomplete type declaration
```

20/2
```
{AI95-00433-01} type Person_Name is access Person;
type Car_Name    is access all Car'Class;
```

21/2
```
{AI95-00433-01} type Car is tagged
   record
      Number  : Integer;
      Owner   : Person_Name;
   end record;
```

22
```
type Person(Sex : Gender) is
   record
      Name     : String(1 .. 20);
      Birth    : Date;
      Age      : Integer range 0 .. 130;
      Vehicle  : Car_Name;
      case Sex is
         when M => Wife           : Person_Name(Sex => F);
         when F => Husband        : Person_Name(Sex => M);
      end case;
   end record;
```

23
```
My_Car, Your_Car, Next_Car : Car_Name := new Car;  -- see 4.8
George : Person_Name := new Person(M);
   ...
George.Vehicle := Your_Car;
```

Extensions to Ada 83

23.a {*extensions to Ada 83*} The full_type_declaration that completes an incomplete_type_declaration may have a known_discriminant_part even if the incomplete_type_declaration does not.

23.b/1 A discriminant_constraint may be applied to an incomplete type, even if its completion is deferred to the package body, because there is no "dependent compatibility check" required any more. Of course, the constraint can be specified only if a known_discriminant_part was given in the incomplete_type_declaration. As mentioned in the previous paragraph, that is no longer required even when the full type has discriminants.

Wording Changes from Ada 83

23.c Dereferences producing incomplete types were not explicitly disallowed in RM83, though AI83-00039 indicated that it was not strictly necessary since troublesome cases would result in Constraint_Error at run time, since the access value would necessarily be null. However, this introduces an undesirable implementation burden, as illustrated by Example 4 of AI83-00039:

23.d
```
package Pack is
    type Pri is private;
private
    type Sep;
    type Pri is access Sep;
    X : Pri;
end Pack;
```

23.e
```
package body Pack is -- Could be separately compiled!
    type Sep is ...;
    X := new Sep;
end Pack;
```

23.f
```
pragma Elaborate(Pack);
private package Pack.Child is
    I : Integer := X.all'Size; -- Legal, by AI-00039.
end Pack.Child;
```

23.g Generating code for the above example could be a serious implementation burden, since it would require all aliased objects to store size dope, and for that dope to be in the same format for all kinds of types (or some other equivalently inefficient implementation). On the contrary, most implementations allocate dope differently (or not at all) for different designated subtypes.

Incompatibilities With Ada 95

23.h/2 {*AI95-00326-01*} {*incompatibilities with Ada 95*} It is now illegal to use an incomplete view (type) as the parameter or result of an access-to-subprogram type unless the incomplete view is completed in the same declaration list as the use. This was allowed in Ada 95 for incomplete types where the completion was deferred to the body. By disallowing this rare use of incomplete views, we can allow the use of incomplete views in many more places, which is especially valuable for limited views.

23.i/2 {*AI95-00326-01*} It is now illegal to use an incomplete view (type) in a primitive subprogram of the type unless the incomplete view is completed in the package specification. This was allowed in Ada 95 for incomplete types where the completion was deferred to the body (the use would have to be in an access parameter). This incompatibility was caused by the fix for the hole noted in Legality Rules above.

Extensions to Ada 95

{*AI95-00326-01*} {*extensions to Ada 95*} Tagged incomplete types are new. They are allowed in parameter declarations as well as the usual places, as tagged types are always by-reference types (and thus there can be no code generation issue). 23.j/2

{*AI95-00412-01*} A subtype_declaration can be used to give a new name to an incomplete view of a type. This is valuable to give shorter names to entities imported with a limited_with_clause. 23.k/2

Wording Changes from Ada 95

{*AI95-00326-01*} The description of incomplete types as *incomplete views* is new. Ada 95 defined these as separate types, but neglected to give any rules for matching them with other types. Luckily, implementers did the right thing anyway. This change also makes it easier to describe the meaning of a limited view. 23.l/2

3.10.2 Operations of Access Types

[The attribute Access is used to create access values designating aliased objects and non-intrinsic subprograms. The “accessibility” rules prevent dangling references (in the absence of uses of certain unchecked features — see Section 13).] 1

Language Design Principles

It should be possible for an access value to designate an object declared by an object declaration, or a subcomponent thereof. In implementation terms, this means pointing at stack-allocated and statically allocated data structures. However, dangling references should be prevented, primarily via compile-time rules, so long as features like Unchecked_Access and Unchecked_Deallocation are not used. 1.a

In order to create such access values, we require that the access type be a general access type, that the designated object be aliased, and that the accessibility rules be obeyed. 1.b

Name Resolution Rules

{*AI95-00235-01*} {*expected type (access attribute_reference)* [partial]} For an attribute_reference with attribute_designator Access (or Unchecked_Access — see 13.10), the expected type shall be a single access type *A* such that: 2/2

- {*AI95-00235-01*} *A* is an access-to-object type with designated type *D* and the type of the prefix is *D*'Class or is covered by *D*, or 2.1/2
- {*AI95-00235-01*} *A* is an access-to-subprogram type whose designated profile is type conformant with that of the prefix. 2.2/2

{*AI95-00235-01*} [The prefix of such an attribute_reference is never interpreted as an implicit_dereference or a parameterless function_call (see 4.1.4).] {*expected profile (Access attribute_reference prefix)* [partial]} {*expected type (Access attribute_reference prefix)* [partial]} The designated type or profile of the expected type of the attribute_reference is the expected type or profile for the prefix. 2.3/2

Discussion: Saying that the expected type shall be a "single access type" is our "new" way of saying that the type has to be determinable from context using only the fact that it is an access type. See 4.2 and 8.6. Specifying the expected profile only implies type conformance. The more stringent subtype conformance is required by a Legality Rule. This is the only Resolution Rule that applies to the name in a prefix of an attribute_reference. In all other cases, the name has to be resolved without using context. See 4.1.4. 2.a

{*AI95-00235-01*} Saying “single access type” is a bit of a fudge. Both the context and the prefix may provide both multiple types; “single” only means that a single, specific interpretation must remain after resolution. We say “single” here to trigger the Legality Rules of 8.6. The resolution of an access attribute is similar to that of an assignment_statement. For example: 2.b/2

2.c/2
```
type Int_Ptr is access all Integer;
type Char_Ptr is access all Character;
type Float_Ptr is access all Float;
```

2.d/2
```
function Zap (Val : Int_Ptr) return Float;   -- (1)
function Zap (Val : Float_Ptr) return Float; -- (2)
function Zop return Int_Ptr;  -- (3)
function Zop return Char_Ptr; -- (4)
```

2.e/2
```
Result : Float := Zap (Zop.all'Access); -- Resolves to Zap (1) and Zop (3).
```

Static Semantics

3/2 {*AI95-00162-01*} {*accessibility level*} {*level (accessibility)*} {*deeper (accessibility level)*} {*depth (accessibility level)*} {*dangling references (prevention via accessibility rules)*} {*lifetime*} [The accessibility rules, which prevent dangling references, are written in terms of *accessibility levels*, which reflect the run-time nesting of *masters*. As explained in 7.6.1, a master is the execution of a certain construct, such as a subprogram_body. An accessibility level is *deeper than* another if it is more deeply nested at run time. For example, an object declared local to a called subprogram has a deeper accessibility level than an object declared local to the calling subprogram. The accessibility rules for access types require that the accessibility level of an object designated by an access value be no deeper than that of the access type. This ensures that the object will live at least as long as the access type, which in turn ensures that the access value cannot later designate an object that no longer exists. The Unchecked_Access attribute may be used to circumvent the accessibility rules.]

4 {*statically deeper*} {*deeper (statically)*} [A given accessibility level is said to be *statically deeper* than another if the given level is known at compile time (as defined below) to be deeper than the other for all possible executions. In most cases, accessibility is enforced at compile time by Legality Rules. Run-time accessibility checks are also used, since the Legality Rules do not cover certain cases involving access parameters and generic packages.]

5 Each master, and each entity and view created by it, has an accessibility level:

6 • The accessibility level of a given master is deeper than that of each dynamically enclosing master, and deeper than that of each master upon which the task executing the given master directly depends (see 9.3).

7/2 • {*AI95-00162-01*} {*AI95-00416-01*} An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. A parameter of a master has the same accessibility level as the master.

7.a/2 **Reason:** {*AI95-00416-01*} This rule defines the “normal” accessibility of entities. In the absence of special rules below, we intend for this rule to apply.

7.b/2 **Discussion:** {*AI95-00416-01*} This rule defines the accessibility of all named access types, as well as the accessibility level of all anonymous access types other than those for access parameters and access discriminants. Special rules exist for the accessibility level of such anonymous types. Components, stand-alone objects, and function results whose (anonymous) type is defined by an access_definition have accessibility levels corresponding to named access types defined at the same point.

7.c/2 **Ramification:** {*AI95-00230-01*} Because accessibility level is determined by where the access_definition is elaborated, for a type extension, the anonymous access types of components (other than access discriminants) inherited from the parent have the same accessibility as they did in the parent; those in the extension part have the accessibility determined by the scope where the type extension is declared. Similarly, the types of the non-discriminant access components of a derived untagged type have the same accessibility as they did in the parent.

8 • The accessibility level of a view of an object or subprogram defined by a renaming_declaration is the same as that of the renamed view.

9/2 • {*AI95-00416-01*} The accessibility level of a view conversion, qualified_expression, or parenthesized expression, is the same as that of the operand.

10/2 • {*AI95-00318-02*} {*AI95-00416-01*} The accessibility level of an aggregate or the result of a function call [(or equivalent use of an operator)] that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an aggregate or the result of a function call is that of the innermost master that evaluates the aggregate or function call.

To be honest: {*AI95-00416-01*} The first sentence is talking about a static use of the entire return object - a slice that happens to be the entire return object doesn't count. On the other hand, this is intended to allow parentheses and qualified_expressions. 10.a/2

Ramification: {*AI95-00416-01*} If the function is used as a prefix, the second sentence applies. Similarly, an assignment_statement is not an initialization of an object, so the second sentence applies. 10.b/2

The "innermost master which evaluated the function call" does not include the function call itself (which might be a master). 10.c

We really mean the innermost master here, which could be a very short lifetime. Consider a function call used as a parameter of a procedure call. In this case the innermost master which evaluated the function call is the procedure call. 10.d

- {*AI95-00416-01*} Within a return statement, the accessibility level of the return object is that of the execution of the return statement. If the return statement completes normally by returning from the function, then prior to leaving the function, the accessibility level of the return object changes to be a level determined by the point of call, as does the level of any coextensions (see below) of the return object. 10.1/2

Reason: We define the accessibility level of the return object during the return statement to be that of the return statement itself so that the object may be designated by objects local to the return statement, but not by objects outside the return statement. In addition, the intent is that the return object gets finalized if the return statement ends without actually returning (for example, due to propagating an exception, or a goto). For a normal return, of course, no finalization is done before returning. 10.d.1/2

- The accessibility level of a derived access type is the same as that of its ultimate ancestor. 11
- {*AI95-00230-01*} The accessibility level of the anonymous access type defined by an access_definition of an object_renaming_declaration is the same as that of the renamed view. 11.1/2
- {*AI95-00230-01*} {*AI95-00416-01*} The accessibility level of the anonymous access type of an access discriminant in the subtype_indication or qualified_expression of an allocator, or in the expression or return_subtype_indication of a return statement is determined as follows: 12/2
  - If the value of the access discriminant is determined by a discriminant_association in a subtype_indication, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null); 12.1/2

  Discussion: This deals with the following cases, when they occur in the context of an allocator or return statement: 12.a/2

    - An extension_aggregate where the ancestor_part is a subtype_mark denoting a constrained subtype; 12.b/2
    - An uninitialized allocator where the subtype_indication defines a constrained subtype; 12.c/2
    - A discriminant of an object with a constrained nominal subtype, including constrained components, the result of calling a function with a constrained result subtype, the dereference of an access-to-constrained subtype, etc. 12.d/2

  - If the value of the access discriminant is determined by a record_component_association in an aggregate, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null); 12.2/2

  Discussion: In this bullet, the aggregate has to occur in the context of an allocator or return statement, while the subtype_indication of the previous bullet can occur anywhere (it doesn't have to be directly given in the allocator or return statement). 12.e/2

  - In other cases, where the value of the access discriminant is determined by an object with an unconstrained nominal subtype, the accessibility level of the object. 12.3/2

  Discussion: {*AI95-00416-01*} In other words, if you know the value of the discriminant for an allocator or return statement from a discriminant constraint or an aggregate component association, then that determines the accessibility level; if you don't know it, then it is based on the object itself. 12.e.1/2

- {*AI95-00416-01*} The accessibility level of the anonymous access type of an access discriminant in any other context is that of the enclosing object. 12.4/2

13/2 • {*AI95-00162-01*} {*AI95-00254-01*} The accessibility level of the anonymous access type of an access parameter specifying an access-to-object type is the same as that of the view designated by the actual.

13.1/2 • {*AI95-00254-01*} The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is deeper than that of any master; all such anonymous access types have this same level.

13.a/2 **Reason:** {*downward closure*} {*closure (downward)*} These represent "downward closures" and thus require passing of static links or global display information (along with generic sharing information if the implementation does sharing) along with the address of the subprogram. We must prevent conversions of these to types with "normal" accessibility, as those typically don't include the extra information needed to make a call.

14/2 • {*AI95-00416-01*} The accessibility level of an object created by an allocator is the same as that of the access type, except for an allocator of an anonymous access type that defines the value of an access parameter or an access discriminant. For an allocator defining the value of an access parameter, the accessibility level is that of the innermost master of the call. For one defining an access discriminant, the accessibility level is determined as follows:

14.1/2 • {*AI95-00416-01*} for an allocator used to define the constraint in a subtype_declaration, the level of the subtype_declaration;

14.2/2 • {*AI95-00416-01*} for an allocator used to define the constraint in a component_definition, the level of the enclosing type;

14.3/2 • {*AI95-00416-01*} for an allocator used to define the discriminant of an object, the level of the object.

14.4/2 {*AI95-00416-01*} {*coextension (of an object)*} In this last case, the allocated object is said to be a *coextension* of the object whose discriminant designates it, as well as of any object of which the discriminated object is itself a coextension or subcomponent. All coextensions of an object are finalized when the object is finalized (see 7.6.1).

14.a.1/2 **Ramification:** The rules of access discriminants are such that when the space for an object with a coextension is reclaimed, the space for the coextensions can be reclaimed. Hence, there is implementation advice (see 13.11) that an object and its coextensions all be allocated from the same storage pool (or stack frame, in the case of a declared object).

15 • The accessibility level of a view of an object or subprogram denoted by a dereference of an access value is the same as that of the access type.

16 • The accessibility level of a component, protected subprogram, or entry of (a view of) a composite object is the same as that of (the view of) the composite object.

16.1/2 {*AI95-00416-01*} In the above rules, the operand of a view conversion, parenthesized expression or qualified_expression is considered to be used in a context if the view conversion, parenthesized expression or qualified_expression itself is used in that context.

17 {*statically deeper*} {*deeper (statically)*} One accessibility level is defined to be *statically deeper* than another in the following cases:

18 • For a master that is statically nested within another master, the accessibility level of the inner master is statically deeper than that of the outer master.

18.a **To be honest:** Strictly speaking, this should talk about the *constructs* (such as subprogram_bodies) being statically nested within one another; the masters are really the *executions* of those constructs.

18.b **To be honest:** If a given accessibility level is statically deeper than another, then each level defined to be the same as the given level is statically deeper than each level defined to be the same as the other level.

18.1/2 • {*AI95-00254-01*} The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is statically deeper than that of any master; all such anonymous access types have this same level.

Ramification: This rule means that it is illegal to convert an access parameter specifying an access to subprogram to a named access to subprogram type, but it is allowed to pass such an access parameter to another access parameter (the implicit conversion's accessibility will succeed). 18.c/2

- {*AI95-00254-01*} The statically deeper relationship does not apply to the accessibility level of the anonymous type of an access parameter specifying an access-to-object type; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other. 19/2
- For determining whether one level is statically deeper than another when within a generic package body, the generic package is presumed to be instantiated at the same level as where it was declared; run-time checks are needed in the case of more deeply nested instantiations. 20
- For determining whether one level is statically deeper than another when within the declarative region of a type_declaration, the current instance of the type is presumed to be an object created at a deeper level than that of the type. 21

Ramification: In other words, the rules are checked at compile time of the type_declaration, in an assume-the-worst manner. 21.a

{*library level*} {*level (library)*} The accessibility level of all library units is called the *library level*; a library-level declaration or entity is one whose accessibility level is the library level. 22

Ramification: Library_unit_declarations are library level. Nested declarations are library level if they are nested only within packages (possibly more than one), and not within subprograms, tasks, etc. 22.a

To be honest: The definition of the accessibility level of the anonymous type of an access parameter specifying an access-to-object type cheats a bit, since it refers to the view designated by the actual, but access values designate objects, not views of objects. What we really mean is the view that "would be" denoted by an expression "X.**all**", where X is the actual, even though such an expression is a figment of our imagination. The definition is intended to be equivalent to the following more verbose version: The accessibility level of the anonymous type of an access parameter is as follows: 22.b/2

- if the actual is an expression of a named access type — the accessibility level of that type; 22.c
- if the actual is an allocator — the accessibility level of the execution of the called subprogram; 22.d
- if the actual is a reference to the Access attribute — the accessibility level of the view denoted by the prefix; 22.e/1
- if the actual is a reference to the Unchecked_Access attribute — library accessibility level; 22.f
- if the actual is an access parameter — the accessibility level of its type. 22.g

Note that the allocator case is explicitly mentioned in the RM95, because otherwise the definition would be circular: the level of the anonymous type is that of the view designated by the actual, which is that of the access type. 22.h

Discussion: A deeper accessibility level implies a shorter maximum lifetime. Hence, when a rule requires X to have a level that is "not deeper than" Y's level, this requires that X has a lifetime at least as long as Y. (We say "maximum lifetime" here, because the accessibility level really represents an upper bound on the lifetime; an object created by an allocator can have its lifetime prematurely ended by an instance of Unchecked_Deallocation.) 22.i

Package elaborations are not masters, and are therefore invisible to the accessibility rules: an object declared immediately within a package has the same accessibility level as an object declared immediately within the declarative region containing the package. This is true even in the body of a package; it jibes with the fact that objects declared in a package_body live as long as objects declared outside the package, even though the body objects are not visible outside the package. 22.j

Note that the level of the *view* denoted by X.**all** can be different from the level of the *object* denoted by X.**all**. The former is determined by the type of X; the latter is determined either by the type of the allocator, or by the master in which the object was declared. The former is used in several Legality Rules and run-time checks; the latter is used to define when X.**all** gets finalized. The level of a view reflects what we can conservatively "know" about the object of that view; for example, due to type_conversions, an access value might designate an object that was allocated by an allocator for a different access type. 22.k

Similarly, the level of the view denoted by X.**all**.Comp can be different from the level of the object denoted by X.**all**.Comp. 22.l

If Y is statically deeper than X, this implies that Y will be (dynamically) deeper than X in all possible executions. 22.m

Most accessibility checking is done at compile time; the rules are stated in terms of "statically deeper than". The exceptions are: 22.n

22.o/2
- Checks involving access parameters of an access-to-object type. The fact that “statically deeper than” is not defined for the anonymous access type of an access parameter implies that any rule saying “shall not be statically deeper than” does not apply to such a type, nor to anything defined to have “the same” level as such a type.

22.p
- Checks involving entities and views within generic packages. This is because an instantiation can be at a level that is more deeply nested than the generic package itself. In implementations that use a macro-expansion model of generics, these violations can be detected at macro-expansion time. For implementations that share generics, run-time code is needed to detect the error.

22.q/2
- {*AI95-00318-02*} {*AI95-00344-01*} {*AI95-00416-01*} Checks during function return and allocators, for nested type extensions and access discriminants.

22.r Note that run-time checks are not required for access discriminants, because their accessibility is determined statically by the accessibility level of the enclosing object.

22.s/2 The accessibility level of the result object of a function reflects the time when that object will be finalized; we don't allow pointers to the object to survive beyond that time.

22.t We sometimes use the terms “accessible” and “inaccessible” to mean that something has an accessibility level that is not deeper, or deeper, respectively, than something else.

22.u/2 **Implementation Note:** {*AI95-00318-02*} {*AI95-00344-01*} {*AI95-00416-01*} If an accessibility Legality Rule is satisfied, then the corresponding run-time check (if any) cannot fail (and a reasonable implementation will not generate any checking code) unless one of the cases requiring run-time checks mentioned previously is involved.

22.v Accessibility levels are defined in terms of the relations “the same as” and “deeper than”. To make the discussion more concrete, we can assign actual numbers to each level. Here, we assume that library-level accessibility is level 0, and each level defined as “deeper than” is one level deeper. Thus, a subprogram directly called from the environment task (such as the main subprogram) would be at level 1, and so on.

22.w/2 Accessibility is not enforced at compile time for access parameters of an access-to-object type. The “obvious” implementation of the run-time checks would be inefficient, and would involve distributed overhead; therefore, an efficient method is given below. The “obvious” implementation would be to pass the level of the caller at each subprogram call, task creation, etc. This level would be incremented by 1 for each dynamically nested master. An Accessibility_Check would be implemented as a simple comparison — checking that X is not deeper than Y would involve checking that X <= Y.

22.x A more efficient method is based on passing *static* nesting levels (within constructs that correspond at run time to masters — packages don't count). Whenever an access parameter is passed, an implicit extra parameter is passed with it. The extra parameter represents (in an indirect way) the accessibility level of the anonymous access type, and, therefore, the level of the view denoted by a dereference of the access parameter. This is analogous to the implicit “Constrained” bit associated with certain formal parameters of an unconstrained but definite composite subtype. In this method, we avoid distributed overhead: it is not necessary to pass any extra information to subprograms that have no access parameters. For anything other than an access parameter and its anonymous type, the static nesting level is known at compile time, and is defined analogously to the RM95 definition of accessibility level (e.g. derived access types get their nesting level from their parent). Checking “not deeper than” is a "<=" test on the levels.

22.y/2 For each access parameter of an access-to-object type, the static depth passed depends on the actual, as follows:

22.z
- If the actual is an expression of a named access type, pass the static nesting level of that type.

22.aa
- If the actual is an allocator, pass the static nesting level of the caller, plus one.

22.bb/1
- If the actual is a reference to the Access attribute, pass the level of the view denoted by the prefix.

22.cc
- If the actual is a reference to the Unchecked_Access attribute, pass 0 (the library accessibility level).

22.dd/2
- If the actual is an access parameter of an access-to-object type, usually just pass along the level passed in. However, if the static nesting level of the formal (access) parameter is greater than the static nesting level of the actual (access) parameter, the level to be passed is the minimum of the static nesting level of the access parameter and the actual level passed in.

22.ee/2 For the Accessibility_Check associated with a type_conversion of an access parameter of an access-to-object type of a given subprogram to a named access type, if the target type is statically nested within the subprogram, do nothing; the check can't fail in this case. Otherwise, check that the value passed in is <= the static nesting depth of the target type. The other Accessibility_Checks are handled in a similar manner.

22.ff This method, using statically known values most of the time, is efficient, and, more importantly, avoids distributed overhead.

Discussion: Examples of accessibility: 22.gg

22.hh
```
package body Lib_Unit is
    type T is tagged ...;
    type A0 is access all T;
    Global: A0 := ...;
    procedure P(X: T) is
        Y: aliased T;
        type A1 is access all T;
        Ptr0: A0 := Global; -- OK.
        Ptr1: A1 := X'Access; -- OK.
    begin
        Ptr1 := Y'Access; -- OK;
        Ptr0 := A0(Ptr1); -- Illegal type conversion!
        Ptr0 := X'Access; -- Illegal reference to Access attribute!
        Ptr0 := Y'Access; -- Illegal reference to Access attribute!
        Global := Ptr0; -- OK.
    end P;
end Lib_Unit;
```

The above illegal statements are illegal because the accessibility level of X and Y are statically deeper than the accessibility level of A0. In every possible execution of any program including this library unit, if P is called, the accessibility level of X will be (dynamically) deeper than that of A0. Note that the accessibility levels of X and Y are the same. 22.ii

Here's an example involving access parameters of an access-to-object type: 22.jj/2

22.kk
```
procedure Main is
    type Level_1_Type is access all Integer;
```

22.ll
```
    procedure P(X: access Integer) is
        type Nested_Type is access all Integer;
    begin
        ... Nested_Type(X) ... -- (1)
        ... Level_1_Type(X) ... -- (2)
    end P;
```

22.mm
```
    procedure Q(X: access Integer) is
        procedure Nested(X: access Integer) is
        begin
            P(X);
        end Nested;
    begin
        Nested(X);
    end Q;
```

22.nn
```
    procedure R is
        Level_2: aliased Integer;
    begin
        Q(Level_2'Access); -- (3)
    end R;
```

22.oo
```
    Level_1: aliased Integer;
begin
    Q(Level_1'Access); -- (4)
    R;
end Main;
```

The run-time Accessibility_Check at (1) can never fail, and no code should be generated to check it. The check at (2) will fail when called from (3), but not when called from (4). 22.pp

Within a type_declaration, the rules are checked in an assume-the-worst manner. For example: 22.qq

22.rr
```
package P is
    type Int_Ptr is access all Integer;
    type Rec(D: access Integer) is limited private;
private
    type Rec_Ptr is access all Rec;
    function F(X: Rec_Ptr) return Boolean;
    function G(X: access Rec) return Boolean;
    type Rec(D: access Integer) is
        record
            C1: Int_Ptr := Int_Ptr(D); -- Illegal!
            C2: Rec_Ptr := Rec'Access; -- Illegal!
            C3: Boolean := F(Rec'Access); -- Illegal!
            C4: Boolean := G(Rec'Access);
        end record;
end P;
```

22.ss C1, C2, and C3 are all illegal, because one might declare an object of type Rec at a more deeply nested place than the declaration of the type. C4 is legal, but the accessibility level of the object will be passed to function G, and constraint checks within G will prevent it from doing any evil deeds.

22.tt Note that we cannot defer the checks on C1, C2, and C3 until compile-time of the object creation, because that would cause violation of the privacy of private parts. Furthermore, the problems might occur within a task or protected body, which the compiler can't see while compiling an object creation.

23 The following attribute is defined for a prefix X that denotes an aliased view of an object:

24/1 X'Access {*8652/0010*} {*AI95-00127-01*} X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. {*Unchecked_Access attribute: See also Access attribute*} X shall denote an aliased view of an object[, including possibly the current instance (see 8.6) of a limited type within its definition, or a formal parameter or generic formal object of a tagged type]. The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X'Access is the general access type *A* with designated type *D*:

25
- If *A* is an access-to-variable type, then the view shall be a variable; [on the other hand, if *A* is an access-to-constant type, the view may be either a constant or a variable.]

25.a **Discussion:** The current instance of a limited type is considered a variable.

26/2
- {*AI95-00363-01*} The view shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is constrained by its initial value.

26.a **Discussion:** This restriction is intended to be similar to the restriction on renaming discriminant-dependent subcomponents.

26.b **Reason:** This prevents references to subcomponents that might disappear or move or change constraints after creating the reference.

26.c **Implementation Note:** There was some thought to making this restriction more stringent, roughly: "X shall not denote a subcomponent of a variable with discriminant-dependent subcomponents, if the nominal subtype of the variable is an unconstrained definite subtype." This was because in some implementations, it is not just the discriminant-dependent subcomponents that might move as the result of an assignment that changed the discriminants of the enclosing object. However, it was decided not to make this change because a reasonable implementation strategy was identified to avoid such problems, as follows:

26.d
- Place non-discriminant-dependent components with any aliased parts at offsets preceding any discriminant-dependent components in a discriminated record type with defaulted discriminants.

26.e
- Preallocate the maximum space for unconstrained discriminated variables with aliased subcomponents, rather than allocating the initial size and moving them to a larger (heap-resident) place if they grow as the result of an assignment.

26.f Note that for objects of a by-reference type, it is not an error for a programmer to take advantage of the fact that such objects are passed by reference. Therefore, the above approach is also necessary for discriminated record types with components of a by-reference type.

26.g To make the above strategy work, it is important that a component of a derived type is defined to be discriminant-dependent if it is inherited and the parent subtype constraint is defined in terms of a discriminant of the derived type (see 3.7).

26.h/2 **To be honest:** {*AI95-00363-01*} If X is a subcomponent that depends on discriminants, and the subcomponent is a dereference of a general access type whose designated type is unconstrained and whose discriminants have defaults, the attribute is illegal. Such a general access type can designate an unconstrained (stack) object. Since such a type might not designate an object constrained by its initial value, the 'Access is illegal — the rule says “is” constrained by its initial value, not “might be” constrained by its initial value. No other interpretation makes sense, as we can't have legality depending on something (which object is designated) that is not known at compile-time, and we surely can't allow this for unconstrained objects. The wording of the rule should be much clearer on this point, but this was discovered after the completion of Amendment 1 when it was too late to fix it.

27/2
- {*8652/0010*} {*AI95-00127-01*} {*AI95-00363-01*} If *A* is a named access type and *D* is a tagged type, then the type of the view shall be covered by *D*; if *A* is anonymous

and *D* is tagged, then the type of the view shall be either *D*'Class or a type covered by *D*; if *D* is untagged, then the type of the view shall be *D*, and either:

- {*AI95-00363-01*} the designated subtype of *A* shall statically match the nominal subtype of the view; or{*statically matching (required)* [partial]} 27.1/2
- {*AI95-00363-01*} *D* shall be discriminated in its full view and unconstrained in any partial view, and the designated subtype of *A* shall be unconstrained. 27.2/2

Implementation Note: This ensures that the dope for an aliased array object can always be stored contiguous with it, but need not be if its nominal subtype is constrained. 27.a

Ramification: {*8652/0010*} {*AI95-00127-01*} An access attribute can be used as the controlling operand in a dispatching call; see 3.9.2. 27.a.1/1

{*AI95-00363-01*} This does not require that types have a partial view in order to allow an access attribute of an unconstrained discriminated object, only that any partial view that does exist is unconstrained. 27.a.2/2

- The accessibility level of the view shall not be statically deeper than that of the access type *A*. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. {*accessibility rule (Access attribute)* [partial]} {*generic contract issue* [partial]} 28

Ramification: In an instance body, a run-time check applies. 28.a

{*AI95-00230-01*} If *A* is an anonymous access-to-object type of an access parameter, then the view can never have a deeper accessibility level than *A*. The same is true for an anonymous access-to-object type of an access discriminant, except when X'Access is used to initialize an access discriminant of an object created by an allocator. The latter case is illegal if the accessibility level of X is statically deeper than that of the access type of the allocator; a run-time check is needed in the case where the initial value comes from an access parameter. Other anonymous access-to-object types have "normal" accessibility checks. 28.b/2

{*Accessibility_Check* [partial]} {*check, language-defined (Accessibility_Check)*} {*Program_Error (raised by failure of run-time check)*} A check is made that the accessibility level of X is not deeper than that of the access type *A*. If this check fails, Program_Error is raised. 29

Ramification: The check is needed for access parameters of an access-to-object type and in instance bodies. 29.a/2

Implementation Note: This check requires that some indication of lifetime is passed as an implicit parameter along with access parameters of an access-to-object type. No such requirement applies to other anonymous access types, since the checks associated with them are all compile-time checks. 29.b/2

{*implicit subtype conversion (Access attribute)* [partial]} If the nominal subtype of X does not statically match the designated subtype of *A*, a view conversion of X to the designated subtype is evaluated (which might raise Constraint_Error — see 4.6) and the value of X'Access designates that view. 30

The following attribute is defined for a prefix P that denotes a subprogram: 31

P'Access {*AI95-00229-01*} {*AI95-00254-01*} P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. {*accessibility rule (Access attribute)* [partial]} The accessibility level of P shall not be statically deeper than that of *S*. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. The profile of P shall be subtype-conformant with the designated profile of *S*, and shall not be Intrinsic. {*subtype conformance (required)*} If the subprogram denoted by P is declared within a generic unit, and the expression P'Access occurs within the body of that generic unit or within the body of a generic unit declared within the declarative region of the generic unit, then the ultimate ancestor of *S* shall be either a non-formal type declared within the generic unit or an anonymous access type of an access parameter. 32/2

Discussion: {*AI95-00229-01*} The part about generic bodies is worded in terms of the denoted subprogram, not the denoted view; this implies that renaming is invisible to this part of the rule. “Declared within the declarative region of the generic” is referring to child and nested generic units.This rule is partly to prevent contract model problems with 32.a/2 respect to the accessibility rules, and partly to ease shared-generic-body implementations, in which a subprogram declared in an instance needs to have a different calling convention from other subprograms with the same profile.

32.b Overload resolution ensures only that the profile is type-conformant. This rule specifies that subtype conformance is required (which also requires matching calling conventions). P cannot denote an entry because access-to-subprogram types never have the *entry* calling convention. P cannot denote an enumeration literal or an attribute function because these have intrinsic calling conventions.

NOTES

33 84 The Unchecked_Access attribute yields the same result as the Access attribute for objects, but has fewer restrictions (see 13.10). There are other predefined operations that yield access values: an allocator can be used to create an object, and return an access value that designates it (see 4.8); evaluating the literal **null** yields a null access value that designates no entity at all (see 4.2).

34/2 85 {*AI95-00230-01*} {*predefined operations (of an access type)* [partial]} The predefined operations of an access type also include the assignment operation, qualification, and membership tests. Explicit conversion is allowed between general access types with matching designated subtypes; explicit conversion is allowed between access-to-subprogram types with subtype conformant profiles (see 4.6). {*subtype conformance* [partial]} Named access types have predefined equality operators; anonymous access types do not, but they can use the predefined equality operators for *universal_access* (see 4.5.2).

34.a/2 **Reason:** {*AI95-00230-01*} Anonymous access types can use the universal access equality operators declared in Standard, while named access types cannot for compatibility reasons. By not having equality operators for anonymous access types, we eliminate the need to specify exactly where the predefined operators for anonymous access types would be defined, as well as the need for an implementer to insert an implicit declaration for "=", etc. at the appropriate place in their symbol table. Note that ":=", 'Access, and ".**all**" are defined.

35 86 The object or subprogram designated by an access value can be named with a dereference, either an explicit_-dereference or an implicit_dereference. See 4.1.

36 87 A call through the dereference of an access-to-subprogram value is never a dispatching call.

36.a **Proof:** See 3.9.2.

37/2 88 {*AI95-00254-01*} {*downward closure*} {*closure (downward)*} The Access attribute for subprograms and parameters of an anonymous access-to-subprogram type may together be used to implement “downward closures” — that is, to pass a more-nested subprogram as a parameter to a less-nested subprogram, as might be appropriate for an iterator abstraction or numerical integration. Downward closures can also be implemented using generic formal subprograms (see 12.6). Note that Unchecked_Access is not allowed for subprograms.

38 89 Note that using an access-to-class-wide tagged type with a dispatching operation is a potentially more structured alternative to using an access-to-subprogram type.

39 90 An implementation may consider two access-to-subprogram values to be unequal, even though they designate the same subprogram. This might be because one points directly to the subprogram, while the other points to a special prologue that performs an Elaboration_Check and then jumps to the subprogram. See 4.5.2.

39.a **Ramification:** If equality of access-to-subprogram values is important to the logic of a program, a reference to the Access attribute of a subprogram should be evaluated only once and stored in a global constant for subsequent use and equality comparison.

Examples

40 *Example of use of the Access attribute:*

41
```
Martha : Person_Name := new Person(F);       -- see 3.10.1
Cars   : array (1..2) of aliased Car;
   ...
Martha.Vehicle := Cars(1)'Access;
George.Vehicle := Cars(2)'Access;
```

Extensions to Ada 83

41.a {*extensions to Ada 83*} We no longer make things like 'Last and ".component" (basic) operations of an access type that need to be "declared" somewhere. Instead, implicit dereference in a prefix takes care of them all. This means that there should never be a case when X.**all**'Last is legal while X'Last is not. See AI83-00154.

Incompatibilities With Ada 95

41.b/2 {*AI95-00363-01*} {*incompatibilities with Ada 95*} Aliased variables are not necessarily constrained in Ada 2005 (see 3.6). Therefore, a subcomponent of an aliased variable may disappear or change shape, and taking 'Access of such a subcomponent thus is illegal, while the same operation would have been legal in Ada 95. Note that most allocated objects are still constrained by their initial value (see 4.8), and thus legality of 'Access didn't change for them. For example:

41.c/2
```
type T1 (D1 : Boolean := False) is
   record
      case D1 is
         when False =>
            C1 : aliased Integer;
         when True =>
            null;
      end case;
   end record;
type Acc_Int is access all Integer;
```

41.d/2
```
A_T : aliased T1;
Ptr : Acc_Int := A_T.C1'Access; -- Illegal in Ada 2005, legal in Ada 95
A_T := (D1 => True);            -- Raised Constraint_Error in Ada 95, but does not
                                -- in Ada 2005, so Ptr would become invalid when this
                                -- is assigned (thus Ptr is illegal).
```

{*AI95-00363-01*} If a discriminated full type has a partial view (private type) that is constrained, we do not allow 'Access on objects to create a value of an object of an access-to-unconstrained type. Ada 95 allowed this attribute and various access subtypes, requiring that the heap object be constrained and thus making details of the implementation of the private type visible to the client of the private type. See 4.8 for more on this topic. 41.e/2

{*AI95-00229-01*} {*AI95-00254-01*} **Amendment Correction:** Taking 'Access of a subprogram declared in a generic unit in the body of that generic is no longer allowed. Such references can easily be used to create dangling pointers, as Legality Rules are not rechecked in instance bodies. At the same time, the rules were loosened a bit where that is harmless, and also to allow any routine to be passed to an access parameter of an access-to-subprogram type. The now illegal uses of 'Access can almost always be moved to the private part of the generic unit, where they are still legal (and rechecked upon instantiation for possibly dangling pointers). 41.f/2

Extensions to Ada 95

{*8652/0010*} {*AI95-00127-01*} **Corrigendum:** {*extensions to Ada 95*} Access attributes of objects of class-wide types can be used as the controlling parameter in a dispatching calls (see 3.9.2). This was an oversight in Ada 95. 41.g/2

{*AI95-00235-01*} **Amendment Correction:** The type of the prefix can now be used in resolving Access attributes. This allows more uses of the Access attribute to resolve. For example: 41.h/2

41.i/2
```
type Int_Ptr is access all Integer;
type Float_Ptr is access all Float;
```

41.j/2
```
function Zap (Val : Int_Ptr) return Float;
function Zap (Val : Float_Ptr) return Float;
```

41.k/2
```
Value : aliased Integer := 10;
```

41.l/2
```
Result1 : Float := Zap (Value'access); -- Ambiguous in Ada 95; resolves in Ada 2005.
Result2 : Float := Zap (Int_Ptr'(Value'access)); -- Resolves in Ada 95 and Ada 2005.
```

This change is upward compatible; any expression that does not resolve by the new rules would have failed a Legality Rule. 41.m/2

Wording Changes from Ada 95

{*AI95-00162-01*} Adjusted the wording to reflect the fact that expressions and function calls are masters. 41.n/2

{*AI95-00230-01*} {*AI95-00254-01*} {*AI95-00318-02*} {*AI95-00385-01*} {*AI95-00416-01*} Defined the accessibility of the various new kinds and uses of anonymous access types. 41.o/2

3.11 Declarative Parts

[A declarative_part contains declarative_items (possibly none).] 1

Syntax

declarative_part ::= {declarative_item} 2

declarative_item ::=
  basic_declarative_item | body 3

{*8652/0009*} {*AI95-00137-01*} basic_declarative_item ::=
  basic_declaration | aspect_clause | use_clause 4/1

5 body ::= proper_body | body_stub

6 proper_body ::=
  subprogram_body | package_body | task_body | protected_body

Static Semantics

6.1/2 {*AI95-00420-01*} The list of declarative_items of a declarative_part is called the *declaration list* of the declarative_part.{*declaration list (declarative_part)* [partial]}

Dynamic Semantics

7 {*elaboration (declarative_part)* [partial]} The elaboration of a declarative_part consists of the elaboration of the declarative_items, if any, in the order in which they are given in the declarative_part.

8 {*elaborated*} An elaborable construct is in the *elaborated* state after the normal completion of its elaboration. Prior to that, it is *not yet elaborated*.

8.a **Ramification:** The elaborated state is only important for bodies; certain uses of a body raise an exception if the body is not yet elaborated.

8.b Note that "prior" implies before the start of elaboration, as well as during elaboration.

8.c The use of the term "normal completion" implies that if the elaboration propagates an exception or is aborted, the declaration is not elaborated. RM83 missed the aborted case.

9 {*Elaboration_Check* [partial]} {*check, language-defined (Elaboration_Check)*} For a construct that attempts to use a body, a check (Elaboration_Check) is performed, as follows:

10/1 • {*8652/0014*} {*AI95-00064-01*} For a call to a (non-protected) subprogram that has an explicit body, a check is made that the body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

10.a **Discussion:** AI83-00180 specifies that there is no elaboration check for a subprogram defined by a pragma Interface (or equivalently, pragma Import). AI83-00430 specifies that there is no elaboration check for an enumeration literal. AI83-00406 specifies that the evaluation of parameters and the elaboration check occur in an arbitrary order. AI83-00406 applies to generic instantiation as well (see below).

10.a.1/1 {*8652/0014*} {*AI95-00064-01*} A subprogram can be completed by a renaming-as-body, and we need to make an elaboration check on such a body, so we use “body” rather than subprogram_body above.

11 • For a call to a protected operation of a protected type (that has a body — no check is performed if a pragma Import applies to the protected type), a check is made that the protected_body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

11.a **Discussion:** A protected type has only one elaboration “bit,” rather than one for each operation, because one call may result in evaluating the barriers of other entries, and because there are no elaborable declarations between the bodies of the operations. In fact, the elaboration of a protected_body does not elaborate the enclosed bodies, since they are not considered independently elaborable.

11.b Note that there is no elaboration check when calling a task entry. Task entry calls are permitted even before the associated task_body has been seen. Such calls are simply queued until the task is activated and reaches a corresponding accept_statement. We considered a similar rule for protected entries — simply queuing all calls until the protected_body was seen, but felt it was not worth the possible implementation overhead, particularly given that there might be multiple instances of the protected type.

12 • For the activation of a task, a check is made by the activator that the task_body is already elaborated. If two or more tasks are being activated together (see 9.2), as the result of the elaboration of a declarative_part or the initialization for the object created by an allocator, this check is done for all of them before activating any of them.

12.a **Reason:** As specified by AI83-00149, the check is done by the activator, rather than by the task itself. If it were done by the task itself, it would be turned into a Tasking_Error in the activator, and the other tasks would still be activated.

- For the instantiation of a generic unit that has a body, a check is made that this body is already elaborated. This check and the evaluation of any explicit_generic_actual_parameters of the instantiation are done in an arbitrary order. 13

{*Program_Error (raised by failure of run-time check)*} The exception Program_Error is raised if any of these checks fails. 14

Extensions to Ada 83

{*AI95-00114-01*} {*extensions to Ada 83*} The syntax for declarative_part is modified to remove the ordering restrictions of Ada 83; that is, the distinction between basic_declarative_items and later_declarative_items within declarative_parts is removed. This means that things like use_clauses and object_declarations can be freely intermixed with things like bodies. 14.a/2

The syntax rule for proper_body now allows a protected_body, and the rules for elaboration checks now cover calls on protected operations. 14.b

Wording Changes from Ada 83

The syntax rule for later_declarative_item is removed; the syntax rule for declarative_item is new. 14.c

RM83 defines "elaborated" and "not yet elaborated" for declarative_items here, and for other things in 3.1, "Declarations". That's no longer necessary, since these terms are fully defined in 3.1. 14.d

In RM83, all uses of declarative_part are optional (except for the one in block_statement with a **declare**) which is sort of strange, since a declarative_part can be empty, according to the syntax. That is, declarative_parts are sort of "doubly optional". In Ada 95, these declarative_parts are always required (but can still be empty). To simplify description, we go further and say (see 5.6, "Block Statements") that a block_statement without an explicit declarative_part is equivalent to one with an empty one. 14.e

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Changed representation clauses to aspect clauses to reflect that they are used for more than just representation. 14.f/2

{*8652/0014*} {*AI95-00064-01*} **Corrigendum:** Clarified that the elaboration check applies to all kinds of subprogram bodies. 14.g/2

{*AI95-00420-01*} Defined "declaration list" to avoid confusion for various rules. Other kinds of declaration list are defined elsewhere. 14.h/2

3.11.1 Completions of Declarations

{*8652/0014*} {*AI95-00064-01*} Declarations sometimes come in two parts. {*requires a completion*} A declaration that requires a second part is said to *require completion*. {*completion (compile-time concept)*} The second part is called the *completion* of the declaration (and of the entity declared), and is either another declaration, a body, or a pragma. A {*body*} *body* is a body, an entry_body, or a renaming-as-body (see 8.5.4). 1/1

Discussion: Throughout the RM95, there are rules about completions that define the following: 1.a

- Which declarations require a corresponding completion. 1.b
- Which constructs can only serve as the completion of a declaration. 1.c
- Where the completion of a declaration is allowed to be. 1.d
- What kinds of completions are allowed to correspond to each kind of declaration that allows one. 1.e

Don't confuse this compile-time concept with the run-time concept of completion defined in 7.6.1. 1.f

Note that the declaration of a private type (if limited) can be completed with the declaration of a task type, which is then completed with a body. Thus, a declaration can actually come in *three* parts. 1.g

{*AI95-00217-06*} In Ada 2005 the limited view of the package contains an incomplete view of the private type, so we can have *four* parts now. 1.h/2

Name Resolution Rules

2 A construct that can be a completion is interpreted as the completion of a prior declaration only if:

3 • The declaration and the completion occur immediately within the same declarative region;

4 • The defining name or defining_program_unit_name in the completion is the same as in the declaration, or in the case of a pragma, the pragma applies to the declaration;

5 • If the declaration is overloadable, then the completion either has a type-conformant profile, or is a pragma. {*type conformance (required)*}

Legality Rules

6 An implicit declaration shall not have a completion. {*requires a completion* [distributed]} For any explicit declaration that is specified to *require completion*, there shall be a corresponding explicit completion.

6.a.1/2 **To be honest:** {*AI95-00217-06*} The implicit declarations occurring in a limited view do have a completion (the explicit declaration occurring in the full view) but that's a special case, since the implicit declarations are actually built from the explicit ones. So they do not *require* a completion, they have one by *fiat*.

6.a **Discussion:** The implicit declarations of predefined operators are not allowed to have a completion. Enumeration literals, although they are subprograms, are not allowed to have a corresponding subprogram_body. That's because the completion rules are described in terms of constructs (subprogram_declarations) and not entities (subprograms). When a completion is required, it has to be explicit; the implicit null package_body that Section 7 talks about cannot serve as the completion of a package_declaration if a completion is required.

7 At most one completion is allowed for a given declaration. Additional requirements on completions appear where each kind of completion is defined.

7.a **Ramification:** A subunit is not a completion; the stub is.

7.b If the completion of a declaration is also a declaration, then *that* declaration might have a completion, too. For example, a limited private type can be completed with a task type, which can then be completed with a task body. This is not a violation of the “at most one completion” rule.

8 {*completely defined*} A type is *completely defined* at a place that is after its full type definition (if it has one) and after all of its subcomponent types are completely defined. A type shall be completely defined before it is frozen (see 13.14 and 7.3).

8.a **Reason:** Index types are always completely defined — no need to mention them. There is no way for a completely defined type to depend on the value of a (still) deferred constant.

NOTES

9 91 Completions are in principle allowed for any kind of explicit declaration. However, for some kinds of declaration, the only allowed completion is a pragma Import, and implementations are not required to support pragma Import for every kind of entity.

9.a **Discussion:** In fact, we expect that implementations will *not* support pragma Import of things like types — it's hard to even define the semantics of what it would mean. Therefore, in practice, *not* every explicit declaration can have a completion. In any case, if an implementation chooses to support pragma Import for, say, types, it can place whatever restrictions on the feature it wants to. For example, it might want the pragma to be a freezing point for the type.

10 92 There are rules that prevent premature uses of declarations that have a corresponding completion. The Elaboration_Checks of 3.11 prevent such uses at run time for subprograms, protected operations, tasks, and generic units. The rules of 13.14, “Freezing Rules” prevent, at compile time, premature uses of other entities such as private types and deferred constants.

Wording Changes from Ada 83

10.a This subclause is new. It is intended to cover all kinds of completions of declarations, be they a body for a spec, a full type for an incomplete or private type, a full constant declaration for a deferred constant declaration, or a pragma Import for any kind of entity.

Wording Changes from Ada 95

10.b/2 {*8652/0014*} {*AI95-00064-01*} **Corrigendum:** Added a definition of *body*, which is different than body or **body**.

Section 4: Names and Expressions

1 [The rules applicable to the different forms of name and expression, and to their evaluation, are given in this section.]

4.1 Names

1 [Names can denote declared entities, whether declared explicitly or implicitly (see 3.1). Names can also denote objects or subprograms designated by access values; the results of type_conversions or function_calls; subcomponents and slices of objects and values; protected subprograms, single entries, entry families, and entries in families of entries. Finally, names can denote attributes of any of the foregoing.]

Syntax

```
name ::=                                                    2
    direct_name            | explicit_dereference
  | indexed_component      | slice
  | selected_component     | attribute_reference
  | type_conversion        | function_call
  | character_literal
```

3 direct_name ::= identifier | operator_symbol

3.a/2 **Discussion:** {*AI95-00114-01*} character_literal is no longer a direct_name. character_literals are usable even when the corresponding enumeration type declaration is not visible. See 4.2.

4 prefix ::= name | implicit_dereference

5 explicit_dereference ::= name.**all**

6 implicit_dereference ::= name

7/2 [Certain forms of name (indexed_components, selected_components, slices, and attribute_references) include a prefix that is either itself a name that denotes some related entity, or an implicit_dereference of an access value that designates some related entity.]

Name Resolution Rules

8 {*dereference*} {*expected type (dereference name)* [partial]} The name in a *dereference* (either an implicit_dereference or an explicit_dereference) is expected to be of any access type.

Static Semantics

9 {*nominal subtype (associated with a dereference)* [partial]} If the type of the name in a dereference is some access-to-object type *T*, then the dereference denotes a view of an object, the *nominal subtype* of the view being the designated subtype of *T*.

9.a **Ramification:** If the value of the name is the result of an access type conversion, the dereference denotes a view created as part of the conversion. The nominal subtype of the view is not necessarily the same as that used to create the designated object. See 4.6.

9.b **To be honest:** {*nominal subtype (of a name)* [partial]} We sometimes refer to the nominal subtype of a particular kind of name rather than the nominal subtype of the view denoted by the name (presuming the name denotes a view of an object). These two uses of nominal subtype are intended to mean the same thing.

9.c/2 {*AI95-00363-01*} If an allocator for the access-to-object type *T* is one that creates objects that are constrained by their initial value (see 4.8), the subtype of the dereference is constrained even if the designated subtype of *T* is not. We don't want the effect of the dereference to depend on the designated object. This matters because general access-to-unconstrained can designate both allocated objects (which are constrained at birth) and aliased stack objects (which aren't necessarily constrained). This is a wording bug that was discovered after the completion of Amendment 1 when it was too late to fix it; we expect that it will be corrected by an early Ada 2005 AI.

9.d/2 **Implementation Note:** {*AI95-00363-01*} Since we don't depend on whether the designated object is constrained, it is not necessary to include a constrained bit in every object that could be designated by a general access type.

10 {*profile (associated with a dereference)* [partial]} If the type of the name in a dereference is some access-to-subprogram type *S*, then the dereference denotes a view of a subprogram, the *profile* of the view being the designated profile of *S*.

10.a **Ramification:** This means that the formal parameter names and default expressions to be used in a call whose name or prefix is a dereference are those of the designated profile, which need not be the same as those of the subprogram designated by the access value, since 'Access requires only subtype conformance, not full conformance.

Dynamic Semantics

11/2 {*AI95-00415-01*} {*evaluation (name)* [partial]} The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is a direct_name or a character_literal.

12 {*evaluation (name that has a prefix)* [partial]} [The evaluation of a name that has a prefix includes the evaluation of the prefix.] {*evaluation (prefix)* [partial]} The evaluation of a prefix consists of the evaluation of the name or the implicit_dereference. The prefix denotes the entity denoted by the name or the implicit_dereference.

13 {*evaluation (dereference)* [partial]} The evaluation of a dereference consists of the evaluation of the name and the determination of the object or subprogram that is designated by the value of the name. {*Access_Check* [partial]} {*check, language-defined (Access_Check)*} A check is made that the value of the name is not the null access value. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails. The dereference denotes the object or subprogram designated by the value of the name.

Examples

14 *Examples of direct names:*

15
```
Pi      -- the direct name of a number            (see 3.3.2)
Limit   -- the direct name of a constant          (see 3.3.1)
Count   -- the direct name of a scalar variable   (see 3.3.1)
Board   -- the direct name of an array variable   (see 3.6.1)
Matrix  -- the direct name of a type              (see 3.6)
Random  -- the direct name of a function          (see 6.1)
Error   -- the direct name of an exception        (see 11.1)
```

16 *Examples of dereferences:*

17
```
Next_Car.all     --  explicit dereference denoting the object designated by
                 --  the access variable Next_Car (see 3.10.1)
Next_Car.Owner   --  selected component with implicit dereference;
                 --  same as Next_Car.all.Owner
```

Extensions to Ada 83

17.a {*extensions to Ada 83*} Type conversions and function calls are now considered names that denote the result of the operation. In the case of a type conversion used as an actual parameter or that is of a tagged type, the type conversion is considered a variable if the operand is a variable. This simplifies the description of "parameters of the form of a type conversion" as well as better supporting an important OOP paradigm that requires the combination of a conversion from a class-wide type to some specific type followed immediately by component selection. Function calls are considered names so that a type conversion of a function call and the function call itself are treated equivalently in the grammar. A function call is considered the name of a constant, and can be used anywhere such a name is permitted. See 6.5.

17.b/1 Type conversions of a tagged type are permitted anywhere their operand is permitted. That is, if the operand is a variable, then the type conversion can appear on the left-hand side of an assignment_statement. If the operand is an object, then the type conversion can appear in an object renaming or as a prefix. See 4.6.

Wording Changes from Ada 83

{*AI95-00114-01*} Everything of the general syntactic form name(...) is now syntactically a name. In any realistic parser, this would be a necessity since distinguishing among the various name(...) constructs inevitably requires name resolution. In cases where the construct yields a value rather than an object, the name denotes a value rather than an object. Names already denote values in Ada 83 with named numbers, components of the result of a function call, etc. This is partly just a wording change, and partly an extension of functionality (see Extensions heading above). 17.c/2

The syntax rule for direct_name is new. It is used in places where direct visibility is required. It's kind of like Ada 83's simple_name, but simple_name applied to both direct visibility and visibility by selection, and furthermore, it didn't work right for operator_symbols. The syntax rule for simple_name is removed, since its use is covered by a combination of direct_name and selector_name. The syntactic categories direct_name and selector_name are similar; it's mainly the visibility rules that distinguish the two. The introduction of direct_name requires the insertion of one new explicit textual rule: to forbid statement_identifiers from being operator_symbols. This is the only case where the explicit rule is needed, because this is the only case where the declaration of the entity is implicit. For example, there is no need to syntactically forbid (say) “X: "Rem";”, because it is impossible to declare a type whose name is an operator_symbol in the first place. 17.d

The syntax rules for explicit_dereference and implicit_dereference are new; this makes other rules simpler, since dereferencing an access value has substantially different semantics from selected_components. We also use name instead of prefix in the explicit_dereference rule since that seems clearer. Note that these rules rely on the fact that function calls are now names, so we don't need to use prefix to allow functions calls in front of .**all**. 17.e

Discussion: Actually, it would be reasonable to allow any primary in front of .**all**, since only the value is needed, but that would be a bit radical. 17.f

We no longer use the term *appropriate for a type* since we now describe the semantics of a prefix in terms of implicit dereference. 17.g

4.1.1 Indexed Components

[An indexed_component denotes either a component of an array or an entry in a family of entries. {*array indexing: See indexed_component*}] 1

Syntax

indexed_component ::= prefix(expression {, expression}) 2

Name Resolution Rules

The prefix of an indexed_component with a given number of expressions shall resolve to denote an array (after any implicit dereference) with the corresponding number of index positions, or shall resolve to denote an entry family of a task or protected object (in which case there shall be only one expression). 3

{*expected type (indexed_component expression)* [partial]} The expected type for each expression is the corresponding index type. 4

Static Semantics

When the prefix denotes an array, the indexed_component denotes the component of the array with the specified index value(s). {*nominal subtype (associated with an indexed_component)* [partial]} The nominal subtype of the indexed_component is the component subtype of the array type. 5

When the prefix denotes an entry family, the indexed_component denotes the individual entry of the entry family with the specified index value. 6

Dynamic Semantics

{*evaluation (indexed_component)* [partial]} For the evaluation of an indexed_component, the prefix and the expressions are evaluated in an arbitrary order. The value of each expression is converted to the corresponding index type. {*implicit subtype conversion (array index)* [partial]} {*Index_Check* [partial]} {*check, language-defined (Index_Check)*} A check is made that each index value belongs to the corresponding index 7 range of the array or entry family denoted by the prefix. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails.

Examples

8 *Examples of indexed components:*

9
```
My_Schedule(Sat)      -- a component of a one-dimensional array  (see 3.6.1)
Page(10)              -- a component of a one-dimensional array  (see 3.6)
Board(M, J + 1)       -- a component of a two-dimensional array  (see 3.6.1)
Page(10)(20)          -- a component of a component              (see 3.6)
Request(Medium)       -- an entry in a family of entries         (see 9.1)
Next_Frame(L)(M, N)   -- a component of a function call          (see 6.1)
```

NOTES

10 1 *Notes on the examples:* Distinct notations are used for components of multidimensional arrays (such as Board) and arrays of arrays (such as Page). The components of an array of arrays are arrays and can therefore be indexed. Thus Page(10)(20) denotes the 20th component of Page(10). In the last example Next_Frame(L) is a function call returning an access value that designates a two-dimensional array.

4.1.2 Slices

1 [{*array slice*} A slice denotes a one-dimensional array formed by a sequence of consecutive components of a one-dimensional array. A slice of a variable is a variable; a slice of a constant is a constant;] a slice of a value is a value.

Syntax

2 slice ::= prefix(discrete_range)

Name Resolution Rules

3 The prefix of a slice shall resolve to denote a one-dimensional array (after any implicit dereference).

4 {*expected type (slice discrete_range)* [partial]} The expected type for the discrete_range of a slice is the index type of the array type.

Static Semantics

5 A slice denotes a one-dimensional array formed by the sequence of consecutive components of the array denoted by the prefix, corresponding to the range of values of the index given by the discrete_range.

6 The type of the slice is that of the prefix. Its bounds are those defined by the discrete_range.

Dynamic Semantics

7 {*evaluation (slice)* [partial]} For the evaluation of a slice, the prefix and the discrete_range are evaluated in an arbitrary order. {*Index_Check* [partial]} {*check, language-defined (Index_Check)*} {*null slice*} If the slice is not a *null slice* (a slice where the discrete_range is a null range), then a check is made that the bounds of the discrete_range belong to the index range of the array denoted by the prefix. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails.

NOTES

8 2 A slice is not permitted as the prefix of an Access attribute_reference, even if the components or the array as a whole are aliased. See 3.10.2.

8.a **Proof:** Slices are not aliased, by 3.10, “Access Types”.

8.b **Reason:** This is to ease implementation of general-access-to-array. If slices were aliased, implementations would need to store array dope with the access values, which is not always desirable given access-to-incomplete types completed in a package body.

9 3 For a one-dimensional array A, the slice A(N .. N) denotes an array that has only one component; its type is the type of A. On the other hand, A(N) denotes a component of the array A and has the corresponding component type.

Examples

10 *Examples of slices:*

11
```
Stars(1 .. 15)          -- a slice of 15 characters         (see 3.6.3)
Page(10 .. 10 + Size)   -- a slice of 1 + Size components   (see 3.6)
Page(L)(A .. B)         -- a slice of the array Page(L)     (see 3.6)
Stars(1 .. 0)           -- a null slice                     (see 3.6.3)
My_Schedule(Weekday)    -- bounds given by subtype          (see 3.6.1 and 3.5.1)
Stars(5 .. 15)(K)       -- same as Stars(K)                 (see 3.6.3)
                        -- provided that K is in 5 .. 15
```

4.1.3 Selected Components

1 [Selected_components are used to denote components (including discriminants), entries, entry families, and protected subprograms; they are also used as expanded names as described below. {*dot selection: See selected_component*}]

Syntax

2 selected_component ::= prefix . selector_name

3 selector_name ::= identifier | character_literal | operator_symbol

Name Resolution Rules

4 {*expanded name*} A selected_component is called an *expanded name* if, according to the visibility rules, at least one possible interpretation of its prefix denotes a package or an enclosing named construct (directly, not through a subprogram_renaming_declaration or generic_renaming_declaration).

4.a **Discussion:** See AI83-00187.

5 A selected_component that is not an expanded name shall resolve to denote one of the following:

5.a **Ramification:** If the prefix of a selected_component denotes an enclosing named construct, then the selected_component is interpreted only as an expanded name, even if the named construct is a function that could be called without parameters.

6 • A component [(including a discriminant)]:

7 The prefix shall resolve to denote an object or value of some non-array composite type (after any implicit dereference). The selector_name shall resolve to denote a discriminant_specification of the type, or, unless the type is a protected type, a component_declaration of the type. The selected_component denotes the corresponding component of the object or value.

7.a/1 **Reason:** The components of a protected object cannot be named except by an expanded name, even from within the corresponding protected body. The protected body may not reference the private components of some arbitrary object of the protected type; the protected body may reference components of the current instance only (by an expanded name or a direct_name).

7.b **Ramification:** Only the discriminants and components visible at the place of the selected_component can be selected, since a selector_name can only denote declarations that are visible (see 8.3).

8 • A single entry, an entry family, or a protected subprogram:

9 The prefix shall resolve to denote an object or value of some task or protected type (after any implicit dereference). The selector_name shall resolve to denote an entry_declaration or subprogram_declaration occurring (implicitly or explicitly) within the visible part of that type. The selected_component denotes the corresponding entry, entry family, or protected subprogram.

9.a **Reason:** This explicitly says “visible part” because even though the body has visibility on the private part, it cannot call the private operations of some arbitrary object of the task or protected type, only those of the current instance (and expanded name notation has to be used for that).

9.1/2 • {*AI95-00252-01*} {*AI95-00407-01*} A view of a subprogram whose first formal parameter is of a tagged type or is an access parameter whose designated type is tagged:

9.2/2 The prefix (after any implicit dereference) shall resolve to denote an object or value of a specific tagged type *T* or class-wide type *T*'Class. The selector_name shall resolve to denote a view of a subprogram declared immediately within the declarative region in which an ancestor of the type *T* is declared. The first formal parameter of the subprogram shall be of type *T*, or a class-wide type that covers *T*, or an access parameter designating one of these types. The designator of the subprogram shall not be the same as that of a component of the tagged type visible at the point of the selected_component. The selected_component denotes a view of this subprogram that omits the first formal parameter. This view is called a *prefixed view* of the subprogram, and the prefix of the selected_component (after any implicit dereference) is called the *prefix* of the prefixed view. {*prefixed view*} {*prefix (of a prefixed view)*}

10 An expanded name shall resolve to denote a declaration that occurs immediately within a named declarative region, as follows:

11 • The prefix shall resolve to denote either a package [(including the current instance of a generic package, or a rename of a package)], or an enclosing named construct.

12 • The selector_name shall resolve to denote a declaration that occurs immediately within the declarative region of the package or enclosing construct [(the declaration shall be visible at the place of the expanded name — see 8.3)]. The expanded name denotes that declaration.

12.a **Ramification:** Hence, a library unit or subunit can use an expanded name to refer to the declarations within the private part of its parent unit, as well as to other children that have been mentioned in with_clauses.

13 • If the prefix does not denote a package, then it shall be a direct_name or an expanded name, and it shall resolve to denote a program unit (other than a package), the current instance of a type, a block_statement, a loop_statement, or an accept_statement (in the case of an accept_-statement or entry_body, no family index is allowed); the expanded name shall occur within the declarative region of this construct. Further, if this construct is a callable construct and the prefix denotes more than one such enclosing callable construct, then the expanded name is ambiguous, independently of the selector_name.

Legality Rules

13.1/2 {*AI95-00252-01*} {*AI95-00407-01*} For a subprogram whose first parameter is an access parameter, the prefix of any prefixed view shall denote an aliased view of an object.

13.2/2 {*AI95-00407-01*} For a subprogram whose first parameter is of mode **in out** or **out**, or of an anonymous access-to-variable type, the prefix of any prefixed view shall denote a variable.

13.a/2 **Reason:** We want calls through a prefixed view and through a normal view to have the same legality. Thus, the implicit 'Access in this new notation needs the same legality check that an explicit 'Access would have. Similarly, we need to prohibit the object from being constant if the first parameter of the subprogram is **in out**, because that is (obviously) prohibited for passing a normal parameter.

Dynamic Semantics

14 {*evaluation (selected_component)* [partial]} The evaluation of a selected_component includes the evaluation of the prefix.

15 {*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} For a selected_component that denotes a component of a variant, a check is made that the values of the discriminants are such that the value or object denoted by the prefix has this component. {*Constraint_Error (raised by failure of run-time check)*} {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised if this check fails.

Examples

Examples of selected components: 16

17/2
```
{AI95-00252-01} {AI95-00407-01}   Tomorrow.Month      -- a record component  (see 3.8)
  Next_Car.Owner       -- a record component                   (see 3.10.1)
  Next_Car.Owner.Age   -- a record component                   (see 3.10.1)
                       -- the previous two lines involve implicit dereferences
  Writer.Unit          -- a record component (a discriminant)  (see 3.8.1)
  Min_Cell(H).Value    -- a record component of the result     (see 6.1)
                       -- of the function call Min_Cell(H)
  Cashier.Append       -- a prefixed view of a procedure       (see 3.9.4)
  Control.Seize        -- an entry of a protected object       (see 9.4)
  Pool(K).Write        -- an entry of the task Pool(K)         (see 9.4)
```

Examples of expanded names: 18

19
```
  Key_Manager."<"       -- an operator of the visible part of a package  (see 7.3.1)
  Dot_Product.Sum       -- a variable declared in a function body        (see 6.1)
  Buffer.Pool           -- a variable declared in a protected unit       (see 9.11)
  Buffer.Read           -- an entry of a protected unit                  (see 9.11)
  Swap.Temp             -- a variable declared in a block statement      (see 5.6)
  Standard.Boolean      -- the name of a predefined type                 (see A.1)
```

Extensions to Ada 83

{*extensions to Ada 83*} We now allow an expanded name to use a prefix that denotes a rename of a package, even if the selector is for an entity local to the body or private part of the package, so long as the entity is visible at the place of the reference. This eliminates a preexisting anomaly where references in a package body may refer to declarations of its visible part but not those of its private part or body when the prefix is a rename of the package. 19.a

Wording Changes from Ada 83

The syntax rule for selector_name is new. It is used in places where visibility, but not necessarily direct visibility, is required. See 4.1, “Names” for more information. 19.b

The description of dereferencing an access type has been moved to 4.1, “Names”; name.**all** is no longer considered a selected_component. 19.c

The rules have been restated to be consistent with our new terminology, to accommodate class-wide types, etc. 19.d

Extensions to Ada 95

{*AI95-00252-01*} {*extensions to Ada 95*} The prefixed view notation for tagged objects is new. This provides a similar notation to that used in other popular languages, and also reduces the need for use_clauses. This is sometimes known as “distinguished receiver notation”. {*distinguished receiver notation*} 19.e/2

Given the following definitions for a tagged type T: 19.f/2

19.g/2
```
procedure Do_Something (Obj : in out T; Count : in Natural);
procedure Do_Something_Else (Obj : access T; Flag : in Boolean);
My_Object : aliased T;
```

the following calls are equivalent: 19.h/2

19.i/2
```
Do_Something (My_Object, Count => 10);
My_Object.Do_Something (Count => 10);
```

as are the following calls: 19.j/2

19.k/2
```
Do_Something_Else (My_Object'Access, Flag => True);
My_Object.Do_Something_Else (Flag => True);
```

4.1.4 Attributes

{*attribute*} [An *attribute* is a characteristic of an entity that can be queried via an attribute_reference or a range_attribute_reference.] 1

Syntax

attribute_reference ::= prefix'attribute_designator 2

3 attribute_designator ::=
    identifier[(*static*_expression)]
  | Access | Delta | Digits

4 range_attribute_reference ::= prefix'range_attribute_designator

5 range_attribute_designator ::= Range[(*static*_expression)]

Name Resolution Rules

6 In an attribute_reference, if the attribute_designator is for an attribute defined for (at least some) objects of an access type, then the prefix is never interpreted as an implicit_dereference; otherwise (and for all range_attribute_references), if the type of the name within the prefix is of an access type, the prefix is interpreted as an implicit_dereference. Similarly, if the attribute_designator is for an attribute defined for (at least some) functions, then the prefix is never interpreted as a parameterless function_call; otherwise (and for all range_attribute_references), if the prefix consists of a name that denotes a function, it is interpreted as a parameterless function_call.

6.a **Discussion:** The first part of this rule is essentially a "preference" against implicit dereference, so that it is possible to ask for, say, 'Size of an access object, without automatically getting the size of the object designated by the access object. This rule applies to 'Access, 'Unchecked_Access, 'Size, and 'Address, and any other attributes that are defined for at least some access objects.

6.b The second part of this rule implies that, for a parameterless function F, F'Address is the address of F, whereas F'Size is the size of the anonymous constant returned by F.

6.c/1 We normally talk in terms of expected type or profile for name resolution rules, but we don't do this for attributes because certain attributes are legal independent of the type or the profile of the prefix.

6.d/2 {*AI95-00114-01*} Other than the rules given above, the Name Resolution Rules for the prefix of each attribute are defined as Name Resolution Rules for that attribute. If no such rules are defined, then no context at all should be used when resolving the prefix. In particular, any knowledge about the kind of entities required must not be used for resolution unless that is required by Name Resolution Rules. This matters in obscure cases; for instance, given the following declarations:

6.e/2
```
function Get_It return Integer is ... -- (1)
function Get_It return Some_Record_Type is ... -- (2)
```

6.f/2 the following attribute_reference cannot be resolved and is illegal:

6.g/2
```
if Get_It'Valid then
```

6.h/2 even though the Valid attribute is only defined for objects of scalar types, and thus cannot be applied to the result of function (2). That information cannot be used to resolve the prefix. The same would be true if (2) was been a procedure; even though the procedure does not denote an object, the attribute_reference is still illegal.

7 {*expected type (attribute_designator expression)* [partial]} {*expected type (range_attribute_designator expression)* [partial]} The expression, if any, in an attribute_designator or range_attribute_designator is expected to be of any integer type.

Legality Rules

8 The expression, if any, in an attribute_designator or range_attribute_designator shall be static.

Static Semantics

9 An attribute_reference denotes a value, an object, a subprogram, or some other kind of program entity.

9.a **Ramification:** The attributes defined by the language are summarized in Annex K. Implementations can define additional attributes.

10 [A range_attribute_reference X'Range(N) is equivalent to the range X'First(N) .. X'Last(N), except that the prefix is only evaluated once. Similarly, X'Range is equivalent to X'First .. X'Last, except that the prefix is only evaluated once.]

Dynamic Semantics

{*evaluation (attribute_reference)* [partial]} {*evaluation (range_attribute_reference)* [partial]} The evaluation of an attribute_reference (or range_attribute_reference) consists of the evaluation of the prefix. 11

Implementation Permissions

{*8652/0015*} {*AI95-00093-01*} An implementation may provide implementation-defined attributes; the identifier for an implementation-defined attribute shall differ from those of the language-defined attributes unless supplied for compatibility with a previous edition of this International Standard. 12/1

Implementation defined: Implementation-defined attributes. 12.a

Ramification: They cannot be reserved words because reserved words are not legal identifiers. 12.b

The semantics of implementation-defined attributes, and any associated rules, are, of course, implementation defined. For example, the implementation defines whether a given implementation-defined attribute can be used in a static expression. 12.c

{*8652/0015*} {*AI95-00093-01*} Implementations are allowed to support the Small attribute for floating types, as this was defined in Ada 83, even though the name would conflict with a language-defined attribute. 12.c.1/1

NOTES

4 Attributes are defined throughout this International Standard, and are summarized in Annex K. 13

5 {*AI95-00235*} In general, the name in a prefix of an attribute_reference (or a range_attribute_reference) has to be resolved without using any context. However, in the case of the Access attribute, the expected type for the attribute_reference has to be a single access type, and the resolution of the name can use the fact that the type of the object or the profile of the callable entity denoted by the prefix has to match the designated type or be type conformant with the designated profile of the access type. {*type conformance (required)*} 14/2

Proof: {*AI95-00235*} In the general case, there is no “expected type” for the prefix of an attribute_reference. In the special case of 'Access, there is an “expected type” or “expected profile” for the prefix. 14.a/2

Reason: 'Access is a special case, because without it, it would be very difficult to take 'Access of an overloaded subprogram. 14.b

Examples

Examples of attributes: 15

16
```
Color'First          -- minimum value of the enumeration type Color  (see 3.5.1)
Rainbow'Base'First   -- same as Color'First                          (see 3.5.1)
Real'Digits          -- precision of the type Real                   (see 3.5.7)
Board'Last(2)        -- upper bound of the second dimension of Board (see 3.6.1)
Board'Range(1)       -- index range of the first dimension of Board  (see 3.6.1)
Pool(K)'Terminated   -- True if task Pool(K) is terminated           (see 9.1)
Date'Size            -- number of bits for records of type Date      (see 3.8)
Message'Address      -- address of the record variable Message       (see 3.7.1)
```

Extensions to Ada 83

{*extensions to Ada 83*} We now uniformly treat X'Range as X'First..X'Last, allowing its use with scalar subtypes. 16.a

We allow any integer type in the *static*_expression of an attribute designator, not just a value of *universal_integer*. The preference rules ensure upward compatibility. 16.b

Wording Changes from Ada 83

We use the syntactic category attribute_reference rather than simply "attribute" to avoid confusing the name of something with the thing itself. 16.c

The syntax rule for attribute_reference now uses identifier instead of simple_name, because attribute identifiers are not required to follow the normal visibility rules. 16.d

We now separate attribute_reference from range_attribute_reference, and enumerate the reserved words that are legal attribute or range attribute designators. We do this because identifier no longer includes reserved words. 16.e

The Ada 95 name resolution rules are a bit more explicit than in Ada 83. The Ada 83 rule said that the "meaning of the prefix of an attribute must be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute." That isn't quite right since the meaning even in Ada 83 embodies whether or not the 16.f prefix is interpreted as a parameterless function call, and in Ada 95, it also embodies whether or not the prefix is interpreted as an implicit_dereference. So the attribute designator does make a difference — just not much.

16.g Note however that if the attribute designator is Access, it makes a big difference in the interpretation of the prefix (see 3.10.2).

Wording Changes from Ada 95

16.h/2 {*8652/0015*} {*AI95-00093-01*} **Corrigendum:** The wording was changed to allow implementations to continue to implement the Ada 83 Small attribute. This was always intended to be allowed.

16.i/2 {*AI95-00235-01*} The note about resolving prefixes of attributes was updated to reflect that the prefix of an Access attribute now has an expected type (see 3.10.2).

4.2 Literals

1 [{*literal*} A *literal* represents a value literally, that is, by means of notation suited to its kind.] A literal is either a numeric_literal, a character_literal, the literal **null**, or a string_literal. {*constant: See also literal*}

1.a **Discussion:** An enumeration literal that is an identifier rather than a character_literal is not considered a *literal* in the above sense, because it involves no special notation "suited to its kind." It might more properly be called an enumeration_identifier, except for historical reasons.

Name Resolution Rules

2/2 *This paragraph was deleted.*{*AI95-00230-01*}

3 {*expected type (character_literal)* [partial]} {*expected profile (character_literal)* [partial]} For a name that consists of a character_literal, either its expected type shall be a single character type, in which case it is interpreted as a parameterless function_call that yields the corresponding value of the character type, or its expected profile shall correspond to a parameterless function with a character result type, in which case it is interpreted as the name of the corresponding parameterless function declared as part of the character type's definition (see 3.5.1). In either case, the character_literal denotes the enumeration_literal_specification.

3.a **Discussion:** See 4.1.3 for the resolution rules for a selector_name that is a character_literal.

4 {*expected type (string_literal)* [partial]} The expected type for a primary that is a string_literal shall be a single string type.

Legality Rules

5 A character_literal that is a name shall correspond to a defining_character_literal of the expected type, or of the result type of the expected profile.

6 For each character of a string_literal with a given expected string type, there shall be a corresponding defining_character_literal of the component type of the expected string type.

7/2 *This paragraph was deleted.*{*AI95-00230-01*} {*AI95-00231-01*}

Static Semantics

8/2 {*AI95-00230-01*} An integer literal is of type *universal_integer*. A real literal is of type *universal_real*. The literal **null** is of type *universal_access*.

Dynamic Semantics

9 {*evaluation (numeric literal)* [partial]} {*evaluation (null literal)* [partial]} {*null access value*} {*null pointer: See null access value*} The evaluation of a numeric literal, or the literal **null**, yields the represented value.

10 {*evaluation (string_literal)* [partial]} The evaluation of a string_literal that is a primary yields an array value containing the value of each character of the sequence of characters of the string_literal, as defined in 2.6.

The bounds of this array value are determined according to the rules for positional_array_aggregates (see 4.3.3), except that for a null string literal, the upper bound is the predecessor of the lower bound.

11 {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} For the evaluation of a string_literal of type *T*, a check is made that the value of each character of the string_literal belongs to the component subtype of *T*. For the evaluation of a null string literal, a check is made that its lower bound is greater than the lower bound of the base range of the index type. {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised if either of these checks fails.

11.a **Ramification:** The checks on the characters need not involve more than two checks altogether, since one need only check the characters of the string with the lowest and highest position numbers against the range of the component subtype.

NOTES

12 6 Enumeration literals that are identifiers rather than character_literals follow the normal rules for identifiers when used in a name (see 4.1 and 4.1.3). Character_literals used as selector_names follow the normal rules for expanded names (see 4.1.3).

Examples

13 *Examples of literals:*

14
```
3.14159_26536    -- a real literal
1_345           -- an integer literal
'A'             -- a character literal
"Some Text"     -- a string literal
```

Incompatibilities With Ada 83

14.a {*incompatibilities with Ada 83*} Because character_literals are now treated like other literals, in that they are resolved using context rather than depending on direct visibility, additional qualification might be necessary when passing a character_literal to an overloaded subprogram.

Extensions to Ada 83

14.b {*extensions to Ada 83*} Character_literals are now treated analogously to **null** and string_literals, in that they are resolved using context, rather than their content; the declaration of the corresponding defining_character_literal need not be directly visible.

Wording Changes from Ada 83

14.c Name Resolution rules for enumeration literals that are not character_literals are not included anymore, since they are neither syntactically nor semantically "literals" but are rather names of parameterless functions.

Extensions to Ada 95

14.d/2 {*AI95-00230-01*} {*AI95-00231-01*} {*extensions to Ada 95*} **Null** now has type *universal_access*, which is similar to other literals. **Null** can be used with anonymous access types.

4.3 Aggregates

1 [{*aggregate*} An *aggregate* combines component values into a composite value of an array type, record type, or record extension.] {*literal: See also aggregate*}

Syntax

2 aggregate ::= record_aggregate | extension_aggregate | array_aggregate

Name Resolution Rules

3/2 {*AI95-00287-01*} {*expected type (aggregate)* [partial]} The expected type for an aggregate shall be a single array type, record type, or record extension.

3.a **Discussion:** See 8.6, “The Context of Overload Resolution” for the meaning of “shall be a single ... type.”

Legality Rules

4 An aggregate shall not be of a class-wide type.

4.a **Ramification:** When the expected type in some context is class-wide, an aggregate has to be explicitly qualified by the specific type of value to be created, so that the expected type for the aggregate itself is specific.

4.b **Discussion:** We used to disallow aggregates of a type with unknown discriminants. However, that was unnecessarily restrictive in the case of an extension aggregate, and irrelevant to a record aggregate (since a type that is legal for a record aggregate could not possibly have unknown discriminants) and to an array aggregate (the only specific types that can have unknown discriminants are private types, private extensions, and types derived from them).

Dynamic Semantics

5 {*evaluation (aggregate)* [partial]} For the evaluation of an aggregate, an anonymous object is created and values for the components or ancestor part are obtained (as described in the subsequent subclause for each kind of the aggregate) and assigned into the corresponding components or ancestor part of the anonymous object. {*assignment operation (during evaluation of an aggregate)*} Obtaining the values and the assignments occur in an arbitrary order. The value of the aggregate is the value of this object.

5.a **Discussion:** The ancestor part is the set of components inherited from the ancestor type. The syntactic category ancestor_part is the expression or subtype_mark that specifies how the ancestor part of the anonymous object should be initialized.

5.b **Ramification:** The assignment operations do the necessary value adjustment, as described in 7.6. Note that the value as a whole is not adjusted — just the subcomponents (and ancestor part, if any). 7.6 also describes when this anonymous object is finalized.

5.c If the ancestor_part is a subtype_mark the Initialize procedure for the ancestor type is applied to the ancestor part after default-initializing it, unless the procedure is abstract, as described in 7.6. The Adjust procedure for the ancestor type is not called in this case, since there is no assignment to the ancestor part as a whole.

6 {*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} If an aggregate is of a tagged type, a check is made that its value belongs to the first subtype of the type. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails.

6.a **Ramification:** This check ensures that no values of a tagged type are ever outside the first subtype, as required for inherited dispatching operations to work properly (see 3.4). This check will always succeed if the first subtype is unconstrained. This check is not extended to untagged types to preserve upward compatibility.

Extensions to Ada 83

6.b {*extensions to Ada 83*} We now allow extension_aggregates.

Wording Changes from Ada 83

6.c We have adopted new wording for expressing the rule that the type of an aggregate shall be determinable from the outside, though using the fact that it is nonlimited record (extension) or array.

6.d An aggregate now creates an anonymous object. This is necessary so that controlled types will work (see 7.6).

Incompatibilities With Ada 95

6.e/2 {*AI95-00287-01*} {*incompatibilities with Ada 95*} In Ada 95, a limited type is not considered when resolving an aggregate. Since Ada 2005 now allows limited aggregates, we can have incompatibilities. For example:

6.f/2
```
type Lim is limited
   record
      Comp: Integer;
   end record;
```

6.g/2
```
type Not_Lim is
   record
      Comp: Integer;
   end record;
```

6.h/2
```
procedure P(X: Lim);
procedure P(X: Not_Lim);
```

6.i/2
```
P((Comp => 123)); -- Illegal in Ada 2005, legal in Ada 95
```

The call to P is ambiguous in Ada 2005, while it would not be ambiguous in Ada 95 as the aggregate could not have a limited type. Qualifying the aggregate will eliminate any ambiguity. This construction would be rather confusing to a maintenance programmer, so it should be avoided, and thus we expect it to be rare. 6.j/2

Extensions to Ada 95

{*AI95-00287-01*} {*extensions to Ada 95*} Aggregates can be of a limited type. 6.k/2

4.3.1 Record Aggregates

[In a record_aggregate, a value is specified for each component of the record or record extension value, using either a named or a positional association.] 1

Syntax

record_aggregate ::= (record_component_association_list) 2

record_component_association_list ::= 3
  record_component_association {, record_component_association}
 | **null record**

{*AI95-00287-01*} record_component_association ::= 4/2
  [component_choice_list =>] expression
 | component_choice_list => <>

component_choice_list ::= 5
  *component*_selector_name {| *component*_selector_name}
 | **others**

{*named component association*} A record_component_association is a *named component association* if it has a component_choice_list; {*positional component association*} otherwise, it is a *positional component association*. Any positional component associations shall precede any named component associations. If there is a named association with a component_choice_list of **others**, it shall come last. 6

Discussion: These rules were implied by the BNF in an early version of the RM9X, but it made the grammar harder to read, and was inconsistent with how we handle discriminant constraints. Note that for array aggregates we still express some of the rules in the grammar, but array aggregates are significantly different because an array aggregate is either all positional (with a possible **others** at the end), or all named. 6.a

In the record_component_association_list for a record_aggregate, if there is only one association, it shall be a named association. 7

Reason: Otherwise the construct would be interpreted as a parenthesized expression. This is considered a syntax rule, since it is relevant to overload resolution. We choose not to express it with BNF so we can share the definition of record_component_association_list in both record_aggregate and extension_aggregate. 7.a

Ramification: The record_component_association_list of an extension_aggregate does not have such a restriction. 7.b

Name Resolution Rules

{*AI95-00287-01*} {*expected type (record_aggregate)* [partial]} The expected type for a record_aggregate shall be a single record type or record extension. 8/2

Ramification: This rule is used to resolve whether an aggregate is an array_aggregate or a record_aggregate. The presence of a **with** is used to resolve between a record_aggregate and an extension_aggregate. 8.a

{*needed component (record_aggregate record_component_association_list)*} For the record_component_association_list of a record_aggregate, all components of the composite value defined by the aggregate are *needed*[; for the association list of an extension_aggregate, only those components not determined by the ancestor expression or subtype are needed (see 4.3.2).] Each selector_name in a record_component_association shall denote a needed component [(including possibly a discriminant)]. 9

9.a **Ramification:** For the association list of a record_aggregate, "needed components" includes every component of the composite value, but does not include those in unchosen variants (see AI83-309). If there are variants, then the value specified for the discriminant that governs them determines which variant is chosen, and hence which components are needed.

9.b If an extension defines a new known_discriminant_part, then all of its discriminants are needed in the component association list of an extension aggregate for that type, even if the discriminants have the same names and types as discriminants of the type of the ancestor expression. This is necessary to ensure that the positions in the record_component_association_list are well defined, and that discriminants that govern variant_parts can be given by static expressions.

10 {*expected type (record_component_association expression)* [partial]} The expected type for the expression of a record_component_association is the type of the *associated* component(s); {*associated components (of a record_component_association)*} the associated component(s) are as follows:

11
- For a positional association, the component [(including possibly a discriminant)] in the corresponding relative position (in the declarative region of the type), counting only the needed components;

11.a **Ramification:** This means that for an association list of an extension_aggregate, only noninherited components are counted to determine the position.

12
- For a named association with one or more *component*_selector_names, the named component(s);

13
- For a named association with the reserved word **others**, all needed components that are not associated with some previous association.

Legality Rules

14 If the type of a record_aggregate is a record extension, then it shall be a descendant of a record type, through one or more record extensions (and no private extensions).

15 If there are no components needed in a given record_component_association_list, then the reserved words **null record** shall appear rather than a list of record_component_associations.

15.a **Ramification:** For example, "(**null record**)" is a record_aggregate for a null record type. Similarly, "(T'(A) **with null record**)" is an extension_aggregate for a type defined as a null record extension of T.

16/2 {*AI95-00287-01*} Each record_component_association other than an **others** choice with a <> shall have at least one associated component, and each needed component shall be associated with exactly one record_component_association. If a record_component_association with an expression has two or more associated components, all of them shall be of the same type.

16.a/2 **Ramification:** {*AI95-00287-01*} These rules apply to an association with an **others** choice with an expression. An **others** choice with a <> can match zero components or several components with different types..

16.b/2 **Reason:** {*AI95-00287-01*} Without these rules, there would be no way to know what was the expected type for the expression of the association. Note that some of the rules do not apply to <> associations, as we do not need to resolve anything. We allow **others** => <> to match no components as this is similar to array aggregates. That means that (**others** => <>) always represents a default-initialized record or array value.

16.c **Discussion:** AI83-00244 also requires that the expression shall be legal for each associated component. This is because even though two components have the same type, they might have different subtypes. Therefore, the legality of the expression, particularly if it is an array aggregate, might differ depending on the associated component's subtype. However, we have relaxed the rules on array aggregates slightly for Ada 95, so the staticness of an applicable index constraint has no effect on the legality of the array aggregate to which it applies. See 4.3.3. This was the only case (that we know of) where a subtype provided by context affected the legality of an expression.

16.d **Ramification:** The rule that requires at least one associated component for each record_component_association implies that there can be no extra associations for components that don't exist in the composite value, or that are already determined by the ancestor expression or subtype of an extension_aggregate.

16.e The second part of the first sentence ensures that no needed components are left out, nor specified twice.

17 If the components of a variant_part are needed, then the value of a discriminant that governs the variant_part shall be given by a static expression.

17.a **Ramification:** This expression might either be given within the aggregate itself, or in a constraint on the parent subtype in a derived_type_definition for some ancestor of the type of the aggregate.

17.1/2 {*AI95-00287-01*} A record_component_association for a discriminant without a default_expression shall have an expression rather than <>.

17.b/2 **Reason:** A discriminant must always have a defined value, but <> means uninitialized for a discrete type unless the component has a default value.

Dynamic Semantics

18 {*evaluation (record_aggregate)* [partial]} The evaluation of a record_aggregate consists of the evaluation of the record_component_association_list.

19 {*evaluation (record_component_association_list)* [partial]} For the evaluation of a record_component_association_list, any per-object constraints (see 3.8) for components specified in the association list are elaborated and any expressions are evaluated and converted to the subtype of the associated component. {*implicit subtype conversion (expressions in aggregate)* [partial]} Any constraint elaborations and expression evaluations (and conversions) occur in an arbitrary order, except that the expression for a discriminant is evaluated (and converted) prior to the elaboration of any per-object constraint that depends on it, which in turn occurs prior to the evaluation and conversion of the expression for the component with the per-object constraint.

19.a **Ramification:** The conversion in the first rule might raise Constraint_Error.

19.b **Discussion:** This check in the first rule presumably happened as part of the dependent compatibility check in Ada 83.

19.1/2 {*AI95-00287-01*} For a record_component_association with an expression, the expression defines the value for the associated component(s). For a record_component_association with <>, if the component_declaration has a default_expression, that default_expression defines the value for the associated component(s); otherwise, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

20 The expression of a record_component_association is evaluated (and converted) once for each associated component.

NOTES

21 7 For a record_aggregate with positional associations, expressions specifying discriminant values appear first since the known_discriminant_part is given first in the declaration of the type; they have to be in the same order as in the known_discriminant_part.

Examples

22 *Example of a record aggregate with positional associations:*

23
```
(4, July, 1776)                                       -- see 3.8
```

24 *Examples of record aggregates with named associations:*

25
```
(Day => 4, Month => July, Year => 1776)
(Month => July, Day => 4, Year => 1776)
```

26
```
(Disk, Closed, Track => 5, Cylinder => 12)            -- see 3.8.1
(Unit => Disk, Status => Closed, Cylinder => 9, Track => 1)
```

27/2 {*AI95-00287-01*} *Examples of component associations with several choices:*

28
```
(Value => 0, Succ|Pred => new Cell'(0, null, null))     -- see 3.10.1
```

29
```
 -- The allocator is evaluated twice: Succ and Pred designate different cells
```

29.1/2
```
(Value => 0, Succ|Pred => <>)              -- see 3.10.1
```

29.2/2
```
 -- Succ and Pred will be set to null
```

30 *Examples of record aggregates for tagged types (see 3.9 and 3.9.1):*

31
```
Expression'(null record)
Literal'(Value => 0.0)
Painted_Point'(0.0, Pi/2.0, Paint => Red)
```

Extensions to Ada 83

31.a {*extensions to Ada 83*} Null record aggregates may now be specified, via "(**null record**)". However, this syntax is more useful for null record extensions in extension aggregates.

Wording Changes from Ada 83

31.b Various AIs have been incorporated (AI83-00189, AI83-00244, and AI83-00309). In particular, Ada 83 did not explicitly disallow extra values in a record aggregate. Now we do.

Extensions to Ada 95

31.c/2 {*AI95-00287-01*} {*extensions to Ada 95*} <> can be used in place of an expression in a record_aggregate, default initializing the component.

Wording Changes from Ada 95

31.d/2 {*AI95-00287-01*} Limited record_aggregates are allowed (since all kinds of aggregates can now be limited, see 4.3).

4.3.2 Extension Aggregates

1 [An extension_aggregate specifies a value for a type that is a record extension by specifying a value or subtype for an ancestor of the type, followed by associations for any components not determined by the ancestor_part.]

Language Design Principles

1.a The model underlying this syntax is that a record extension can also be viewed as a regular record type with an ancestor "prefix." The record_component_association_list corresponds to exactly what would be needed if there were no ancestor/prefix type. The ancestor_part determines the value of the ancestor/prefix.

Syntax

2 extension_aggregate ::=
   (ancestor_part **with** record_component_association_list)

3 ancestor_part ::= expression | subtype_mark

Name Resolution Rules

4/2 {*AI95-00287-01*} {*expected type (extension_aggregate)* [partial]} The expected type for an extension_aggregate shall be a single type that is a record extension. {*expected type (extension_aggregate ancestor expression)* [partial]} If the ancestor_part is an expression, it is expected to be of any tagged type.

4.a **Reason:** We could have made the expected type *T*'Class where *T* is the ultimate ancestor of the type of the aggregate, or we could have made it even more specific than that. However, if the overload resolution rules get too complicated, the implementation gets more difficult and it becomes harder to produce good error messages.

Legality Rules

5/2 {*AI95-00306-01*} If the ancestor_part is a subtype_mark, it shall denote a specific tagged subtype. If the ancestor_part is an expression, it shall not be dynamically tagged. The type of the extension_aggregate shall be derived from the type of the ancestor_part, through one or more record extensions (and no private extensions).

5.a/2 **Reason:** {*AI95-00306-01*} The expression cannot be dynamically tagged to prevent implicit "truncation" of a dynamically-tagged value to the specific ancestor type. This is similar to the rules in 3.9.2.

Static Semantics

{*needed component (extension_aggregate record_component_association_list)*} For the record_component_association_list of an extension_aggregate, the only components *needed* are those of the composite value defined by the aggregate that are not inherited from the type of the ancestor_part, plus any inherited discriminants if the ancestor_part is a subtype_mark that denotes an unconstrained subtype. 6

Dynamic Semantics

{*evaluation (extension_aggregate)* [partial]} For the evaluation of an extension_aggregate, the record_component_association_list is evaluated. If the ancestor_part is an expression, it is also evaluated; if the ancestor_part is a subtype_mark, the components of the value of the aggregate not given by the record_component_association_list are initialized by default as for an object of the ancestor type. Any implicit initializations or evaluations are performed in an arbitrary order, except that the expression for a discriminant is evaluated prior to any other evaluation or initialization that depends on it. 7

{*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} If the type of the ancestor_part has discriminants that are not inherited by the type of the extension_aggregate, then, unless the ancestor_part is a subtype_mark that denotes an unconstrained subtype, a check is made that each discriminant of the ancestor has the value specified for a corresponding discriminant, either in the record_component_association_list, or in the derived_type_definition for some ancestor of the type of the extension_aggregate. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails. 8

Ramification: Corresponding and specified discriminants are defined in 3.7. The rules requiring static compatibility between new discriminants of a derived type and the parent discriminant(s) they constrain ensure that at most one check is required per discriminant of the ancestor expression. 8.a

NOTES

8 If all components of the value of the extension_aggregate are determined by the ancestor_part, then the record_component_association_list is required to be simply **null record**. 9

9 If the ancestor_part is a subtype_mark, then its type can be abstract. If its type is controlled, then as the last step of evaluating the aggregate, the Initialize procedure of the ancestor type is called, unless the Initialize procedure is abstract (see 7.6). 10

Examples

Examples of extension aggregates (for types defined in 3.9.1): 11

12
```
Painted_Point'(Point with Red)
(Point'(P) with Paint => Black)
```

13
```
(Expression with Left => 1.2, Right => 3.4)
Addition'(Binop with null record)
             -- presuming Binop is of type Binary_Operation
```

Extensions to Ada 83

{*extensions to Ada 83*} The extension aggregate syntax is new. 13.a

Incompatibilities With Ada 95

{*AI95-00306-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** Eliminated implicit "truncation" of a dynamically tagged value when it is used as an ancestor expression. If an aggregate includes such an expression, it is illegal in Ada 2005. Such aggregates are thought to be rare; the problem can be fixed with a type conversion to the appropriate specific type if it occurs. 13.b/2

Wording Changes from Ada 95

{*AI95-00287-01*} Limited extension_aggregates are allowed (since all kinds of aggregates can now be limited, see 4.3). 13.c/2

4.3.3 Array Aggregates

1 [In an array_aggregate, a value is specified for each component of an array, either positionally or by its index.] For a positional_array_aggregate, the components are given in increasing-index order, with a final **others**, if any, representing any remaining components. For a named_array_aggregate, the components are identified by the values covered by the discrete_choices.

Language Design Principles

1.a/1 The rules in this subclause are based on terms and rules for discrete_choice_lists defined in 3.8.1, “Variant Parts and Discrete Choices”. For example, the requirements that **others** come last and stand alone are found there.

Syntax

2
```
array_aggregate ::=
  positional_array_aggregate | named_array_aggregate
```

3/2 {*AI95-00287-01*}
```
positional_array_aggregate ::=
    (expression, expression {, expression})
  | (expression {, expression}, others => expression)
  | (expression {, expression}, others => <>)
```

4
```
named_array_aggregate ::=
    (array_component_association {, array_component_association})
```

5/2 {*AI95-00287-01*}
```
array_component_association ::=
    discrete_choice_list => expression
  | discrete_choice_list => <>
```

6 {*n-dimensional array_aggregate*} An *n-dimensional* array_aggregate is one that is written as n levels of nested array_aggregates (or at the bottom level, equivalent string_literals). {*subaggregate (of an array_aggregate)*} For the multidimensional case (n >= 2) the array_aggregates (or equivalent string_literals) at the n–1 lower levels are called *subaggregate*s of the enclosing n-dimensional array_aggregate. {*array component expression*} The expressions of the bottom level subaggregates (or of the array_aggregate itself if one-dimensional) are called the *array component expressions* of the enclosing n-dimensional array_aggregate.

6.a **Ramification:** Subaggregates do not have a type. They correspond to part of an array. For example, with a matrix, a subaggregate would correspond to a single row of the matrix. The definition of "n-dimensional" array_aggregate applies to subaggregates as well as aggregates that have a type.

6.b **To be honest:** {*others choice*} An ***others** choice* is the reserved word **others** as it appears in a positional_array_aggregate or as the discrete_choice of the discrete_choice_list in an array_component_association.

Name Resolution Rules

7/2 {*AI95-00287-01*} {*expected type (array_aggregate)* [partial]} The expected type for an array_aggregate (that is not a subaggregate) shall be a single array type. {*expected type (array_aggregate component expression)* [partial]} The component type of this array type is the expected type for each array component expression of the array_aggregate.

7.a/2 **Ramification:** {*AI95-00287-01*} We already require a single array or record type or record extension for an aggregate. The above rule requiring a single array type (and similar ones for record and extension aggregates) resolves which kind of aggregate you have.

8 {*expected type (array_aggregate discrete_choice)* [partial]} The expected type for each discrete_choice in any discrete_choice_list of a named_array_aggregate is the type of the *corresponding index*; {*corresponding index (for an array_aggregate)*} the corresponding index for an array_aggregate that is not a

subaggregate is the first index of its type; for an (n–m)-dimensional subaggregate within an array_aggregate of an n-dimensional type, the corresponding index is the index in position m+1.

Legality Rules

9 An array_aggregate of an n-dimensional array type shall be written as an n-dimensional array_aggregate.

9.a **Ramification:** In an m-dimensional array_aggregate [(including a subaggregate)], where m >= 2, each of the expressions has to be an (m–1)-dimensional subaggregate.

10 An **others** choice is allowed for an array_aggregate only if an *applicable index constraint* applies to the array_aggregate. {*applicable index constraint*} [An applicable index constraint is a constraint provided by certain contexts where an array_aggregate is permitted that can be used to determine the bounds of the array value specified by the aggregate.] Each of the following contexts (and none other) defines an applicable index constraint:

- 11/2 {*AI95-00318-02*} For an explicit_actual_parameter, an explicit_generic_actual_parameter, the expression of a return statement, the initialization expression in an object_declaration, or a default_expression [(for a parameter or a component)], when the nominal subtype of the corresponding formal parameter, generic formal parameter, function return object, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype;
- 12 For the expression of an assignment_statement where the name denotes an array variable, the applicable index constraint is the constraint of the array variable;

12.a **Reason:** This case is broken out because the constraint comes from the actual subtype of the variable (which is always constrained) rather than its nominal subtype (which might be unconstrained).

- 13 For the operand of a qualified_expression whose subtype_mark denotes a constrained array subtype, the applicable index constraint is the constraint of the subtype;
- 14 For a component expression in an aggregate, if the component's nominal subtype is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

14.a **Discussion:** Here, the array_aggregate with **others** is being used within a larger aggregate.

- 15 For a parenthesized expression, the applicable index constraint is that, if any, defined for the expression.

15.a **Discussion:** RM83 omitted this case, presumably as an oversight. We want to minimize situations where an expression becomes illegal if parenthesized.

16 The applicable index constraint *applies* to an array_aggregate that appears in such a context, as well as to any subaggregates thereof. In the case of an explicit_actual_parameter (or default_expression) for a call on a generic formal subprogram, no applicable index constraint is defined.

16.a **Reason:** This avoids generic contract model problems, because only mode conformance is required when matching actual subprograms with generic formal subprograms.

17 The discrete_choice_list of an array_component_association is allowed to have a discrete_choice that is a nonstatic expression or that is a discrete_range that defines a nonstatic or null range, only if it is the single discrete_choice of its discrete_choice_list, and there is only one array_component_association in the array_aggregate.

17.a **Discussion:** We now allow a nonstatic **others** choice even if there are other array component expressions as well.

18 In a named_array_aggregate with more than one discrete_choice, no two discrete_choices are allowed to cover the same value (see 3.8.1); if there is no **others** choice, the discrete_choices taken together shall exactly cover a contiguous sequence of values of the corresponding index type.

18.a **Ramification:** This implies that each component must be specified exactly once. See AI83-309.

19 A bottom level subaggregate of a multidimensional array_aggregate of a given array type is allowed to be a string_literal only if the component type of the array type is a character type; each character of such a string_literal shall correspond to a defining_character_literal of the component type.

Static Semantics

20 A subaggregate that is a string_literal is equivalent to one that is a positional_array_aggregate of the same length, with each expression being the character_literal for the corresponding character of the string_literal.

Dynamic Semantics

21 {*evaluation (array_aggregate)* [partial]} The evaluation of an array_aggregate of a given array type proceeds in two steps:

22 1. Any discrete_choices of this aggregate and of its subaggregates are evaluated in an arbitrary order, and converted to the corresponding index type; {*implicit subtype conversion (choices of aggregate)* [partial]}

23 2. The array component expressions of the aggregate are evaluated in an arbitrary order and their values are converted to the component subtype of the array type; an array component expression is evaluated once for each associated component. {*implicit subtype conversion (expressions of aggregate)* [partial]}

23.a **Ramification:** Subaggregates are not separately evaluated. The conversion of the value of the component expressions to the component subtype might raise Constraint_Error.

23.1/2 {*AI95-00287-01*} Each expression in an array_component_association defines the value for the associated component(s). For an array_component_association with <>, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

24 {*bounds (of the index range of an array_aggregate)*} The bounds of the index range of an array_aggregate [(including a subaggregate)] are determined as follows:

25 • For an array_aggregate with an **others** choice, the bounds are those of the corresponding index range from the applicable index constraint;

26 • For a positional_array_aggregate [(or equivalent string_literal)] without an **others** choice, the lower bound is that of the corresponding index range in the applicable index constraint, if defined, or that of the corresponding index subtype, if not; in either case, the upper bound is determined from the lower bound and the number of expressions [(or the length of the string_literal)];

27 • For a named_array_aggregate without an **others** choice, the bounds are determined by the smallest and largest index values covered by any discrete_choice_list.

27.a **Reason:** We don't need to say that each index value has to be covered exactly once, since that is a ramification of the general rule on aggregates that each component's value has to be specified exactly once.

28 {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} For an array_aggregate, a check is made that the index range defined by its bounds is compatible with the corresponding index subtype.

28.a **Discussion:** In RM83, this was phrased more explicitly, but once we define "compatibility" between a range and a subtype, it seems to make sense to take advantage of that definition.

28.b **Ramification:** The definition of compatibility handles the special case of a null range, which is always compatible with a subtype. See AI83-00313.

29 {*Index_Check* [partial]} {*check, language-defined (Index_Check)*} For an array_aggregate with an **others** choice, a check is made that no expression is specified for an index value outside the bounds determined by the applicable index constraint.

Discussion: RM83 omitted this case, apparently through an oversight. AI83-00309 defines this as a dynamic check, even though other Ada 83 rules ensured that this check could be performed statically. We now allow an **others** choice to be dynamic, even if it is not the only choice, so this check now needs to be dynamic, in some cases. Also, within a generic unit, this would be a nonstatic check in some cases. 29.a

{*Index_Check* [partial]} {*check, language-defined (Index_Check)*} For a multidimensional array_aggregate, a check is made that all subaggregates that correspond to the same index have the same bounds. 30

Ramification: No array bounds "sliding" is performed on subaggregates. 30.a

Reason: If sliding were performed, it would not be obvious which subaggregate would determine the bounds of the corresponding index. 30.b

{*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised if any of the above checks fail. 31

NOTES

10 In an array_aggregate, positional notation may only be used with two or more expressions; a single expression in parentheses is interpreted as a parenthesized expression. A named_array_aggregate, such as (1 => X), may be used to specify an array with a single component. 32/2

Examples

Examples of array aggregates with positional associations: 33

34
```
(7, 9, 5, 1, 3, 2, 4, 8, 6, 0)
Table'(5, 8, 4, 1, others => 0)  -- see 3.6
```

Examples of array aggregates with named associations: 35

36
```
(1 .. 5 => (1 .. 8 => 0.0))      -- two-dimensional
(1 .. N => new Cell)             -- N new cells, in particular for N = 0
```

37
```
Table'(2 | 4 | 10 => 1, others => 0)
Schedule'(Mon .. Fri => True,  others => False)  -- see 3.6
Schedule'(Wed | Sun  => False, others => True)
Vector'(1 => 2.5)                                -- single-component vector
```

Examples of two-dimensional array aggregates: 38

39
```
-- Three aggregates for the same value of subtype Matrix(1..2,1..3) (see 3.6):
```

40
```
((1.1, 1.2, 1.3), (2.1, 2.2, 2.3))
(1 => (1.1, 1.2, 1.3), 2 => (2.1, 2.2, 2.3))
(1 => (1 => 1.1, 2 => 1.2, 3 => 1.3), 2 => (1 => 2.1, 2 => 2.2, 3 => 2.3))
```

Examples of aggregates as initial values: 41

42
```
A : Table := (7, 9, 5, 1, 3, 2, 4, 8, 6, 0);          -- A(1)=7, A(10)=0
B : Table := (2 | 4 | 10 => 1, others => 0);          -- B(1)=0, B(10)=1
C : constant Matrix := (1 .. 5 => (1 .. 8 => 0.0)); -- C'Last(1)=5, C'Last(2)=8
```

43
```
D : Bit_Vector(M .. N) := (M .. N => True);         -- see 3.6
E : Bit_Vector(M .. N) := (others => True);
F : String(1 .. 1) := (1 => 'F');  -- a one component aggregate: same as "F"
```

{*AI95-00433-01*} *Example of an array aggregate with defaulted others choice and with an applicable index constraint provided by an enclosing record aggregate:* 44/2

45/2
```
Buffer'(Size => 50, Pos => 1, Value => String'('x', others => <>))  -- see 3.7
```

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} In Ada 95, no applicable index constraint is defined for a parameter in a call to a generic formal subprogram; thus, some aggregates that are legal in Ada 83 are illegal in Ada 95. For example: 45.a.1/1

45.a.2/1
```
subtype S3 is String (1 .. 3);
...
generic
   with function F (The_S3 : in S3) return Integer;
package Gp is
   I : constant Integer := F ((1 => '!', others => '?'));
       -- The aggregate is legal in Ada 83, illegal in Ada 95.
end Gp;
```

45.a.3/1 This change eliminates generic contract model problems.

Extensions to Ada 83

45.a {*extensions to Ada 83*} We now allow "named with others" aggregates in all contexts where there is an applicable index constraint, effectively eliminating what was RM83-4.3.2(6). Sliding never occurs on an aggregate with others, because its bounds come from the applicable index constraint, and therefore already match the bounds of the target.

45.b The legality of an **others** choice is no longer affected by the staticness of the applicable index constraint. This substantially simplifies several rules, while being slightly more flexible for the user. It obviates the rulings of AI83-00244 and AI83-00310, while taking advantage of the dynamic nature of the "extra values" check required by AI83-00309.

45.c Named array aggregates are permitted even if the index type is descended from a formal scalar type. See 4.9 and AI83-00190.

Wording Changes from Ada 83

45.d We now separate named and positional array aggregate syntax, since, unlike other aggregates, named and positional associations cannot be mixed in array aggregates (except that an **others** choice is allowed in a positional array aggregate).

45.e We have also reorganized the presentation to handle multidimensional and one-dimensional aggregates more uniformly, and to incorporate the rulings of AI83-00019, AI83-00309, etc.

Extensions to Ada 95

45.f/2 {*AI95-00287-01*} {*extensions to Ada 95*} <> can be used in place of an expression in an array_aggregate, default-initializing the component.

Wording Changes from Ada 95

45.g/2 {*AI95-00287-01*} Limited array_aggregates are allowed (since all kinds of aggregates can now be limited, see 4.3).

45.h/2 {*AI95-00318-02*} Fixed aggregates to use the subtype of the return object of a function, rather than the result subtype, because they can be different for an extended_return_statement, and we want to use the subtype that's explicitly in the code at the point of the expression.

4.4 Expressions

1 {*expression*} An *expression* is a formula that defines the computation or retrieval of a value. In this International Standard, the term “expression” refers to a construct of the syntactic category expression or of any of the other five syntactic categories defined below. {*and operator*} {*operator (and)*} {*or operator*} {*operator (or)*} {*xor operator*} {*operator (xor)*} {*and then (short-circuit control form)*} {*or else (short-circuit control form)*} {*= operator*} {*operator (=)*} {*equal operator*} {*operator (equal)*} {*/= operator*} {*operator (/=)*} {*not equal operator*} {*operator (not equal)*} {*< operator*} {*operator (<)*} {*less than operator*} {*operator (less than)*} {*<= operator*} {*operator (<=)*} {*less than or equal operator*} {*operator (less than or equal)*} {*> operator*} {*operator (>)*} {*greater than operator*} {*operator (greater than)*} {*>= operator*} {*operator (>=)*} {*greater than or equal operator*} {*operator (greater than or equal)*} {*in (membership test)*} {*not in (membership test)*} {*+ operator*} {*operator (+)*} {*plus operator*} {*operator (plus)*} {*- operator*} {*operator (-)*} {*minus operator*} {*operator (minus)*} {*& operator*} {*operator (&)*} {*ampersand operator*} {*operator (ampersand)*} {*concatenation operator*} {*operator (concatenation)*} {*catenation operator: See concatenation operator*} {** operator*} {*operator (*)*} {*multiply operator*} {*operator (multiply)*} {*times operator*} {*operator (times)*} {*/ operator*} {*operator (/)*} {*divide operator*} {*operator (divide)*} {*mod operator*} {*operator (mod)*} {*rem operator*} {*operator (rem)*} {****

operator} {*operator (**)*} {*exponentiation operator*} {*operator (exponentiation)*} {*abs operator*} {*operator (abs)*} {*absolute value*} {*not operator*} {*operator (not)*}

Syntax

```
expression ::=                                                         2
   relation {and relation}    | relation {and then relation}
 | relation {or relation}     | relation {or else relation}
 | relation {xor relation}

relation ::=                                                           3
   simple_expression [relational_operator simple_expression]
 | simple_expression [not] in range
 | simple_expression [not] in subtype_mark

simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}   4

term ::= factor {multiplying_operator factor}                          5

factor ::= primary [** primary] | abs primary | not primary            6

primary ::=                                                            7
  numeric_literal | null | string_literal | aggregate
 | name | qualified_expression | allocator | (expression)
```

Name Resolution Rules

A name used as a primary shall resolve to denote an object or a value. 8

8.a **Discussion:** This replaces RM83-4.4(3). We don't need to mention named numbers explicitly, because the name of a named number denotes a value. We don't need to mention attributes explicitly, because attributes now denote (rather than yield) values in general. Also, the new wording allows attributes that denote objects, which should always have been allowed (in case the implementation chose to have such a thing).

8.b **Reason:** It might seem odd that this is an overload resolution rule, but it is relevant during overload resolution. For example, it helps ensure that a primary that consists of only the identifier of a parameterless function is interpreted as a function_call rather than directly as a direct_name.

Static Semantics

Each expression has a type; it specifies the computation or retrieval of a value of that type. 9

Dynamic Semantics

{*evaluation (primary that is a name)* [partial]} The value of a primary that is a name denoting an object is the value of the object. 10

Implementation Permissions

{*Overflow_Check* [partial]} {*check, language-defined (Overflow_Check)*} {*Constraint_Error (raised by failure of run-time check)*} For the evaluation of a primary that is a name denoting an object of an unconstrained numeric subtype, if the value of the object is outside the base range of its type, the implementation may either raise Constraint_Error or return the value of the object. 11

11.a **Ramification:** This means that if extra-range intermediates are used to hold the value of an object of an unconstrained numeric subtype, a Constraint_Error can be raised on a read of the object, rather than only on an assignment to it. Similarly, it means that computing the value of an object of such a subtype can be deferred until the first read of the object (presuming no side-effects other than failing an Overflow_Check are possible). This permission is over and above that provided by clause 11.6, since this allows the Constraint_Error to move to a different handler.

11.b **Reason:** This permission is intended to allow extra-range registers to be used efficiently to hold parameters and local variables, even if they might need to be transferred into smaller registers for performing certain predefined operations.

11.c **Discussion:** There is no need to mention other kinds of primarys, since any Constraint_Error to be raised can be "charged" to the evaluation of the particular kind of primary.

Examples

12 *Examples of primaries:*

13
```
4.0                  -- real literal
Pi                   -- named number
(1 .. 10 => 0)       -- array aggregate
Sum                  -- variable
Integer'Last         -- attribute
Sine(X)              -- function call
Color'(Blue)         -- qualified expression
Real(M*N)            -- conversion
(Line_Count + 10)    -- parenthesized expression
```

14 *Examples of expressions:*

15/2
```
{AI95-00433-01} Volume                            -- primary
not Destroyed                 -- factor
2*Line_Count                  -- term
-4.0                          -- simple expression
-4.0 + A                      -- simple expression
B**2 - 4.0*A*C                -- simple expression
R*Sin(θ)*Cos(φ)               -- simple expression
Password(1 .. 3) = "Bwv"      -- relation
Count in Small_Int            -- relation
Count not in Small_Int        -- relation
Index = 0 or Item_Hit         -- expression
(Cold and Sunny) or Warm      -- expression (parentheses are required)
A**(B**C)                     -- expression (parentheses are required)
```

Extensions to Ada 83

15.a {*extensions to Ada 83*} In Ada 83, **out** parameters and their nondiscriminant subcomponents are not allowed as primaries. These restrictions are eliminated in Ada 95.

15.b In various contexts throughout the language where Ada 83 syntax rules had simple_expression, the corresponding Ada 95 syntax rule has expression instead. This reflects the inclusion of modular integer types, which makes the logical operators "**and**", "**or**", and "**xor**" more useful in expressions of an integer type. Requiring parentheses to use these operators in such contexts seemed unnecessary and potentially confusing. Note that the bounds of a range still have to be specified by simple_expressions, since otherwise expressions involving membership tests might be ambiguous. Essentially, the operation ".." is of higher precedence than the logical operators, and hence uses of logical operators still have to be parenthesized when used in a bound of a range.

4.5 Operators and Expression Evaluation

1 [{*precedence of operators*} {*operator precedence*} The language defines the following six categories of operators (given in order of increasing precedence). The corresponding operator_symbols, and only those, can be used as designators in declarations of functions for user-defined operators. See 6.6, “Overloading of Operators”.]

Syntax

2 logical_operator ::= **and** | **or** | **xor**

3 relational_operator ::= = | /= | < | <= | > | >=

4 binary_adding_operator ::= + | – | &

5 unary_adding_operator ::= + | –

6 multiplying_operator ::= * | / | **mod** | **rem**

7 highest_precedence_operator ::= ** | **abs** | **not**

7.a **Discussion:** Some of the above syntactic categories are not used in other syntax rules. They are just used for classification. The others are used for both classification and parsing.

Static Semantics

For a sequence of operators of the same precedence level, the operators are associated with their operands in textual order from left to right. Parentheses can be used to impose specific associations. 8

Discussion: The left-associativity is not directly inherent in the grammar of 4.4, though in 1.1.4 the definition of the metasymbols **{}** implies left associativity. So this could be seen as redundant, depending on how literally one interprets the definition of the **{}** metasymbols. 8.a

See the Implementation Permissions below regarding flexibility in reassociating operators of the same precedence. 8.b

{*predefined operator*} {*operator (predefined)*} For each form of type definition, certain of the above operators are *predefined*; that is, they are implicitly declared immediately after the type definition. {*binary operator*} {*operator (binary)*} {*unary operator*} {*operator (unary)*} For each such implicit operator declaration, the parameters are called Left and Right for *binary* operators; the single parameter is called Right for *unary* operators. [An expression of the form X op Y, where op is a binary operator, is equivalent to a function_call of the form "op"(X, Y). An expression of the form op Y, where op is a unary operator, is equivalent to a function_call of the form "op"(Y). The predefined operators and their effects are described in subclauses 4.5.1 through 4.5.6.] 9

Dynamic Semantics

[{*Constraint_Error (raised by failure of run-time check)*} The predefined operations on integer types either yield the mathematically correct result or raise the exception Constraint_Error. For implementations that support the Numerics Annex, the predefined operations on real types yield results whose accuracy is defined in Annex G, or raise the exception Constraint_Error.] 10

To be honest: Predefined operations on real types can “silently” give wrong results when the Machine_Overflows attribute is false, and the computation overflows. 10.a

Implementation Requirements

{*Constraint_Error (raised by failure of run-time check)*} The implementation of a predefined operator that delivers a result of an integer or fixed point type may raise Constraint_Error only if the result is outside the base range of the result type. 11

{*Constraint_Error (raised by failure of run-time check)*} The implementation of a predefined operator that delivers a result of a floating point type may raise Constraint_Error only if the result is outside the safe range of the result type. 12

To be honest: An exception is made for exponentiation by a negative exponent in 4.5.6. 12.a

Implementation Permissions

For a sequence of predefined operators of the same precedence level (and in the absence of parentheses imposing a specific association), an implementation may impose any association of the operators with operands so long as the result produced is an allowed result for the left-to-right association, but ignoring the potential for failure of language-defined checks in either the left-to-right or chosen order of association. 13

Discussion: Note that the permission to reassociate the operands in any way subject to producing a result allowed for the left-to-right association is not much help for most floating point operators, since reassociation may introduce significantly different round-off errors, delivering a result that is outside the model interval for the left-to-right association. Similar problems arise for division with integer or fixed point operands. 13.a

Note that this permission does not apply to user-defined operators. 13.b

NOTES

11 The two operands of an expression of the form X op Y, where op is a binary operator, are evaluated in an arbitrary order, as for any function_call (see 6.4). 14

Examples

15 *Examples of precedence:*

16
```
not Sunny or Warm     -- same as (not Sunny) or Warm
X > 4.0 and Y > 0.0   -- same as (X > 4.0) and (Y > 0.0)
```

17
```
-4.0*A**2             -- same as –(4.0 * (A**2))
abs(1 + A) + B        -- same as (abs (1 + A)) + B
Y**(-3)               -- parentheses are necessary
A / B * C             -- same as (A/B)*C
A + (B + C)           -- evaluate B + C before adding it to A
```

Wording Changes from Ada 83

17.a We don't give a detailed definition of precedence, since it is all implicit in the syntax rules anyway.

17.b The permission to reassociate is moved here from RM83-11.6(5), so it is closer to the rules defining operator association.

4.5.1 Logical Operators and Short-circuit Control Forms

Name Resolution Rules

1 {*short-circuit control form*} {*and then (short-circuit control form)*} {*or else (short-circuit control form)*} An expression consisting of two relations connected by **and then** or **or else** (a *short-circuit control form*) shall resolve to be of some boolean type; {*expected type (short-circuit control form relation)* [partial]} the expected type for both relations is that same boolean type.

1.a **Reason:** This rule is written this way so that overload resolution treats the two operands symmetrically; the resolution of overloading present in either one can benefit from the resolution of the other. Furthermore, the type expected by context can help.

Static Semantics

2 {*logical operator*} {*operator (logical)*} {*and operator*} {*operator (and)*} {*or operator*} {*operator (or)*} {*xor operator*} {*operator (xor)*} The following logical operators are predefined for every boolean type *T*, for every modular type *T*, and for every one-dimensional array type *T* whose component type is a boolean type: {*bit string: See logical operators on boolean arrays*}

3
```
function "and"(Left, Right : T) return T
function "or" (Left, Right : T) return T
function "xor"(Left, Right : T) return T
```

3.a/2 *This paragraph was deleted.*{*AI95-00145-01*}

3.b/2 **Ramification:** {*AI95-00145-01*} For these operators, we are talking about the type without any (interesting) subtype, and not some subtype with a constraint or exclusion. Since it's possible that there is no name for the “uninteresting” subtype, we denote the type with an italicized *T*. This applies to the italicized *T* in many other predefined operators and attributes as well.{*T (italicized)*}

3.c/2 {*AI95-00145-01*} In many cases, there is a subtype with the correct properties available. The italicized *T* means:

3.d/2 • *T*'Base, for scalars;

3.e/2 • the first subtype of *T*, for tagged types;

3.f/2 • a subtype of the type *T* without any constraint or null exclusion, in other cases.

3.g/2 Note that “without a constraint” is not the same as unconstrained. For instance, a record type with no discriminant part is considered constrained; no subtype of it has a constraint, but the subtype is still constrained.

3.h/2 Thus, the last case often is the same as the first subtype of *T*, but that isn't the case for constrained array types (where the correct subtype is unconstrained) and for access types with a null_exclusion (where the correct subtype does not exclude null).

3.i/2 This italicized *T* is used for defining operators and attributes of the language. The meaning is intended to be as described here.

4 For boolean types, the predefined logical operators **and**, **or**, and **xor** perform the conventional operations of conjunction, inclusive disjunction, and exclusive disjunction, respectively.

5 For modular types, the predefined logical operators are defined on a bit-by-bit basis, using the binary representation of the value of the operands to yield a binary representation for the result, where zero represents False and one represents True. If this result is outside the base range of the type, a final subtraction by the modulus is performed to bring the result into the base range of the type.

6 The logical operators on arrays are performed on a component-by-component basis on matching components (as for equality — see 4.5.2), using the predefined logical operator for the component type. The bounds of the resulting array are those of the left operand.

Dynamic Semantics

7 {*evaluation (short-circuit control form)* [partial]} The short-circuit control forms **and then** and **or else** deliver the same result as the corresponding predefined **and** and **or** operators for boolean types, except that the left operand is always evaluated first, and the right operand is not evaluated if the value of the left operand determines the result.

8 {*Length_Check* [partial]} {*check, language-defined (Length_Check)*} For the logical operators on arrays, a check is made that for each component of the left operand there is a matching component of the right operand, and vice versa. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Also, a check is made that each component of the result belongs to the component subtype. {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised if either of the above checks fails.

8.a **Discussion:** The check against the component subtype is per AI83-00535.

NOTES

9 12 The conventional meaning of the logical operators is given by the following truth table:

10

A	B	(A **and** B)	(A **or** B)	(A **xor** B)
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

Examples

11 *Examples of logical operators:*

12
```
Sunny or Warm
Filter(1 .. 10) and Filter(15 .. 24)   -- see 3.6.1
```

13 *Examples of short-circuit control forms:*

14
```
Next_Car.Owner /= null and then Next_Car.Owner.Age > 25   -- see 3.10.1
N = 0 or else A(N) = Hit_Value
```

4.5.2 Relational Operators and Membership Tests

1 [{*relational operator*} {*operator (relational)*} {*comparison operator: See relational operator*} {*equality operator*} {*operator (equality)*} The *equality operators* = (equals) and /= (not equals) are predefined for nonlimited types. {*ordering operator*} {*operator (ordering)*} The other relational_operators are the *ordering operators* < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). {*= operator*} {*operator (=)*} {*equal operator*} {*operator (equal)*} {*/= operator*} {*operator (/=)*} {*not equal operator*} {*operator (not equal)*} {*< operator*} {*operator (<)*} {*less than operator*} {*operator (less than)*} {*<= operator*} {*operator (<=)*} {*less than or equal operator*} {*operator (less than or equal)*} {*> operator*} {*operator (>)*} {*greater than operator*} {*operator (greater than)*} {*>= operator*} {*operator (>=)*} {*greater than or equal operator*} {*operator*

(greater than or equal)} {*discrete array type*} The ordering operators are predefined for scalar types, and for *discrete array types*, that is, one-dimensional array types whose components are of a discrete type.

1.a **Ramification:** The equality operators are not defined for *every* nonlimited type — see below for the exact rule.

2 {*membership test*} {*in (membership test)*} {*not in (membership test)*} A *membership test*, using **in** or **not in**, determines whether or not a value belongs to a given subtype or range, or has a tag that identifies a type that is covered by a given type. Membership tests are allowed for all types.]

Name Resolution Rules

3/2 {*AI95-00251-01*} {*expected type (membership test simple_expression)* [partial]} {*tested type (of a membership test)*} The *tested type* of a membership test is the type of the range or the type determined by the subtype_mark. If the tested type is tagged, then the simple_expression shall resolve to be of a type that is convertible (see 4.6) to the tested type; if untagged, the expected type for the simple_expression is the tested type.

3.a/2 **Reason:** {*AI95-00230-01*} The part of the rule for untagged types is stated in a way that ensures that operands like a string literal are still legal as operands of a membership test.

3.b/2 {*AI95-00251-01*} The significance of "is convertible to" is that we allow the simple_expression to be of any class-wide type that could be converted to the tested type, not just the one rooted at the tested type. This includes any class-wide type that covers the tested type, along with class-wide interfaces in some cases.

Legality Rules

4 For a membership test, if the simple_expression is of a tagged class-wide type, then the tested type shall be (visibly) tagged.

4.a **Ramification:** Untagged types covered by the tagged class-wide type are not permitted. Such types can exist if they are descendants of a private type whose full type is tagged. This rule is intended to avoid confusion since such derivatives don't have their "own" tag, and hence are indistinguishable from one another at run time once converted to a covering class-wide type.

Static Semantics

5 The result type of a membership test is the predefined type Boolean.

6 The equality operators are predefined for every specific type *T* that is not limited, and not an anonymous access type, with the following specifications:

7
```
function "=" (Left, Right : T) return Boolean
function "/="(Left, Right : T) return Boolean
```

7.1/2 {*AI95-00230-01*} The following additional equality operators for the *universal_access* type are declared in package Standard for use with anonymous access types:

7.2/2
```
function "=" (Left, Right : universal_access) return Boolean
function "/="(Left, Right : universal_access) return Boolean
```

8 The ordering operators are predefined for every specific scalar type *T*, and for every discrete array type *T*, with the following specifications:

9
```
function "<" (Left, Right : T) return Boolean
function "<="(Left, Right : T) return Boolean
function ">" (Left, Right : T) return Boolean
function ">="(Left, Right : T) return Boolean
```

Name Resolution Rules

9.1/2 {*AI95-00230-01*} {*AI95-00420-01*} At least one of the operands of an equality operator for *universal_access* shall be of a specific anonymous access type. Unless the predefined equality operator is identified using an expanded name with prefix denoting the package Standard, neither operand shall be of an access-to-object type whose designated type is *D* or *D*'Class, where *D* has a user-defined primitive equality operator such that:

- its result type is Boolean; 9.2/2
- it is declared immediately within the same declaration list as *D*; and 9.3/2
- at least one of its operands is an access parameter with designated type *D*. 9.4/2

Reason: The first sentence prevents compatibility problems by ensuring that these operators are not used for named access types. Also, universal access types do not count for the purposes of this rule. Otherwise, equality expressions like (X = **null**) would be ambiguous for normal access types. 9.a/2

The rest of the rule makes it possible to call (including a dispatching call) user-defined "=" operators for anonymous access-to-object types (they'd be hidden otherwise), and to write user-defined "=" operations for anonymous access types (by making it possible to see the universal operator using the Standard prefix). 9.b/2

Ramification: We don't need a similar rule for anonymous access-to-subprogram types because they can't be primitive for any type. Note that any non-primitive user-defined equality operators still are hidden by the universal operators; they'll have to be called with a package prefix, but they are likely to be very uncommon. 9.c/2

Legality Rules

{*AI95-00230-01*} At least one of the operands of the equality operators for *universal_access* shall be of type *universal_access*, or both shall be of access-to-object types, or both shall be of access-to-subprogram types. Further: 9.5/2

- When both are of access-to-object types, the designated types shall be the same or one shall cover the other, and if the designated types are elementary or array types, then the designated subtypes shall statically match; 9.6/2
- When both are of access-to-subprogram types, the designated profiles shall be subtype conformant. 9.7/2

Reason: We don't want to allow completely arbitrary comparisons, as we don't want to insist that all access types are represented in ways that are convertible to one another. For instance, a compiler could use completely separate address spaces or incompatible representations. Instead, we allow compares if there exists an access parameter to which both operands could be converted. Since the user could write such an subprogram, and any reasonable meaning for "=" would allow using it in such a subprogram, this doesn't impose any further restrictions on Ada implementations. 9.d/2

Dynamic Semantics

For discrete types, the predefined relational operators are defined in terms of corresponding mathematical operations on the position numbers of the values of the operands. 10

For real types, the predefined relational operators are defined in terms of the corresponding mathematical operations on the values of the operands, subject to the accuracy of the type. 11

Ramification: For floating point types, the results of comparing *nearly* equal values depends on the accuracy of the implementation (see G.2.1, “Model of Floating Point Arithmetic” for implementations that support the Numerics Annex). 11.a

Implementation Note: On a machine with signed zeros, if the generated code generates both plus zero and minus zero, plus and minus zero must be equal by the predefined equality operators. 11.b

Two access-to-object values are equal if they designate the same object, or if both are equal to the null value of the access type. 12

Two access-to-subprogram values are equal if they are the result of the same evaluation of an Access attribute_reference, or if both are equal to the null value of the access type. Two access-to-subprogram values are unequal if they designate different subprograms. {*unspecified* [partial]} [It is unspecified whether two access values that designate the same subprogram but are the result of distinct evaluations of Access attribute_references are equal or unequal.] 13

Reason: This allows each Access attribute_reference for a subprogram to designate a distinct “wrapper” subprogram if necessary to support an indirect call. 13.a

14 {*equality operator (special inheritance rule for tagged types)*} For a type extension, predefined equality is defined in terms of the primitive [(possibly user-defined)] equals operator of the parent type and of any tagged components of the extension part, and predefined equality for any other components not inherited from the parent type.

14.a **Ramification:** Two values of a type extension are not equal if there is a variant_part in the extension part and the two values have different variants present. This is a ramification of the requirement that a discriminant governing such a variant_part has to be a "new" discriminant, and so has to be equal in the two values for the values to be equal. Note that variant_parts in the parent part need not match if the primitive equals operator for the parent type considers them equal.

14.b/2 {*AI95-00349-01*} The full type extension's operation is used for a private extension. This follows as only full types have parent types; the type specified in a private extension is an ancestor, but not necessarily the parent type. For instance, in:

14.c/2
```
with Pak1;
package Pak2 is
   type Typ3 is new Pak1.Typ1 with private;
private
   type Typ3 is new Pak1.Typ2 with null record;
end Pak2;
```

14.d/2 the parent type is Pak1.Typ2, not Pak1.Typ1, and the equality operator of Pak1.Typ2 is used to create predefined equality for Typ3.

15 For a private type, if its full type is tagged, predefined equality is defined in terms of the primitive equals operator of the full type; if the full type is untagged, predefined equality for the private type is that of its full type.

16 {*matching components*} For other composite types, the predefined equality operators [(and certain other predefined operations on composite types — see 4.5.1 and 4.6)] are defined in terms of the corresponding operation on *matching components*, defined as follows:

17 • For two composite objects or values of the same non-array type, matching components are those that correspond to the same component_declaration or discriminant_specification;

18 • For two one-dimensional arrays of the same type, matching components are those (if any) whose index values match in the following sense: the lower bounds of the index ranges are defined to match, and the successors of matching indices are defined to match;

19 • For two multidimensional arrays of the same type, matching components are those whose index values match in successive index positions.

20 The analogous definitions apply if the types of the two objects or values are convertible, rather than being the same.

20.a **Discussion:** Ada 83 seems to omit this part of the definition, though it is used in array type conversions. See 4.6.

21 Given the above definition of matching components, the result of the predefined equals operator for composite types (other than for those composite types covered earlier) is defined as follows:

22 • If there are no components, the result is defined to be True;

23 • If there are unmatched components, the result is defined to be False;

24 • Otherwise, the result is defined in terms of the primitive equals operator for any matching tagged components, and the predefined equals for any matching untagged components.

24.a **Reason:** This asymmetry between tagged and untagged components is necessary to preserve upward compatibility and corresponds with the corresponding situation with generics, where the predefined operations "reemerge" in a generic for untagged types, but do not for tagged types. Also, only tagged types support user-defined assignment (see 7.6), so only tagged types can fully handle levels of indirection in the implementation of the type. For untagged types, one reason for a user-defined equals operator might be to allow values with different bounds or discriminants to compare equal in certain cases. When such values are matching components, the bounds or discriminants will necessarily match anyway if the discriminants of the enclosing values match.

Ramification: Two null arrays of the same type are always equal; two null records of the same type are always equal. 24.b

Note that if a composite object has a component of a floating point type, and the floating point type has both a plus and minus zero, which are considered equal by the predefined equality, then a block compare cannot be used for the predefined composite equality. Of course, with user-defined equals operators for tagged components, a block compare breaks down anyway, so this is not the only special case that requires component-by-component comparisons. On a one's complement machine, a similar situation might occur for integer types, since one's complement machines typically have both a plus and minus (integer) zero. 24.c

To be honest: {*AI95-00230-01*} For a component with an anonymous access type, “predefined equality” is that defined for the *universal_access* type (anonymous access types have no equality operators of their own). 24.d/2

For a component with a tagged type *T*, “the primitive equals operator” is the one with two parameters of *T* which returns Boolean. We're not talking about some random other primitive function named "=". 24.e/2

{*8652/0016*} {*AI95-00123-01*} For any composite type, the order in which "=" is called for components is unspecified. Furthermore, if the result can be determined before calling "=" on some components, it is unspecified whether "=" is called on those components.{*Unspecified* [partial]} 24.1/1

The predefined "/=" operator gives the complementary result to the predefined "=" operator. 25

Ramification: Furthermore, if the user defines an "=" operator that returns Boolean, then a "/=" operator is implicitly declared in terms of the user-defined "=" operator so as to give the complementary result. See 6.6. 25.a

{*lexicographic order*} For a discrete array type, the predefined ordering operators correspond to *lexicographic order* using the predefined order relation of the component type: A null array is lexicographically less than any array having at least one component. In the case of nonnull arrays, the left operand is lexicographically less than the right operand if the first component of the left operand is less than that of the right; otherwise the left operand is lexicographically less than the right operand only if their first components are equal and the tail of the left operand is lexicographically less than that of the right (the *tail* consists of the remaining components beyond the first and can be null). 26

{*evaluation (membership test)* [partial]} For the evaluation of a membership test, the simple_expression and the range (if any) are evaluated in an arbitrary order. 27

A membership test using **in** yields the result True if: 28

- The tested type is scalar, and the value of the simple_expression belongs to the given range, or the range of the named subtype; or 29

  Ramification: The scalar membership test only does a range check. It does not perform any other check, such as whether a value falls in a “hole” of a “holey” enumeration type. The Pos attribute function can be used for that purpose. 29.a

  Even though Standard.Float is an unconstrained subtype, the test “X in Float” will still return False (presuming the evaluation of X does not raise Constraint_Error) when X is outside Float'Range. 29.b

- {*AI95-00231-01*} The tested type is not scalar, and the value of the simple_expression satisfies any constraints of the named subtype, and: 30/2
  - {*AI95-00231-01*} if the type of the simple_expression is class-wide, the value has a tag that identifies a type covered by the tested type; 30.1/2

    Ramification: Note that the tag is not checked if the simple_expression is of a specific type. 30.a
  - {*AI95-00231-01*} if the tested type is an access type and the named subtype excludes null, the value of the simple_expression is not null. 30.2/2

Otherwise the test yields the result False. 31

A membership test using **not in** gives the complementary result to the corresponding membership test using **in**. 32

Implementation Requirements

32.1/1 {*8652/0016*} {*AI95-00123-01*} For all nonlimited types declared in language-defined packages, the "=" and "/=" operators of the type shall behave as if they were the predefined equality operators for the purposes of the equality of composite types and generic formal types.

32.a.1/1 **Ramification:** If any language-defined types are implemented with a user-defined "=" operator, then either the full type must be tagged, or the compiler must use “magic” to implement equality for this type. A normal user-defined "=" operator for an untagged type does *not* meet this requirement.

NOTES

33/2 *This paragraph was deleted.*{*AI95-00230-01*}

34 13 If a composite type has components that depend on discriminants, two values of this type have matching components if and only if their discriminants are equal. Two nonnull arrays have matching components if and only if the length of each dimension is the same for both.

Examples

35 *Examples of expressions involving relational operators and membership tests:*

36
```
X /= Y
```

37
```
"" < "A" and "A" < "Aa"      -- True
"Aa" < "B" and "A" < "A  "   -- True
```

38
```
My_Car = null                -- true if My_Car has been set to null (see 3.10.1)
My_Car = Your_Car            -- true if we both share the same car
My_Car.all = Your_Car.all    -- true if the two cars are identical
```

39
```
N not in 1 .. 10             -- range membership test
Today in Mon .. Fri          -- range membership test
Today in Weekday             -- subtype membership test (see 3.5.1)
Archive in Disk_Unit         -- subtype membership test (see 3.8.1)
Tree.all in Addition'Class   -- class membership test (see 3.9.1)
```

Extensions to Ada 83

39.a {*extensions to Ada 83*} Membership tests can be used to test the tag of a class-wide value.

39.b Predefined equality for a composite type is defined in terms of the primitive equals operator for tagged components or the parent part.

Wording Changes from Ada 83

39.c The term “membership test” refers to the relation "X in S" rather to simply the reserved word **in** or **not in**.

39.d We use the term “equality operator” to refer to both the = (equals) and /= (not equals) operators. Ada 83 referred to = as *the* equality operator, and /= as the inequality operator. The new wording is more consistent with the ISO 10646 name for "=" (equals sign) and provides a category similar to “ordering operator” to refer to both = and /=.

39.e We have changed the term “catenate” to “concatenate”.

Extensions to Ada 95

39.f/2 {*AI95-00230-01*} {*AI95-00420-01*} {*extensions to Ada 95*} The *universal_access* equality operators are new. They provide equality operations (most importantly, testing against **null**) for anonymous access types.

Wording Changes from Ada 95

39.g/2 {*8652/0016*} {*AI95-00123-01*} **Corrigendum:** Wording was added to clarify that the order of calls (and whether the calls are made at all) on "=" for components is unspecified. Also clarified that "=" must compose properly for language-defined types.

39.h/2 {*AI95-00251-01*} Memberships were adjusted to allow interfaces which don't cover the tested type, in order to be consistent with type conversions.

4.5.3 Binary Adding Operators

Static Semantics

1 {*binary adding operator*} {*operator (binary adding)*} {*+ operator*} {*operator (+)*} {*plus operator*} {*operator (plus)*} {*- operator*} {*operator (-)*} {*minus operator*} {*operator (minus)*} The binary adding operators + (addition) and – (subtraction) are predefined for every specific numeric type *T* with their conventional meaning. They have the following specifications:

2
```
function "+"(Left, Right : T) return T
function "-"(Left, Right : T) return T
```

3 {*& operator*} {*operator (&)*} {*ampersand operator*} {*operator (ampersand)*} {*concatenation operator*} {*operator (concatenation)*} {*catenation operator: See concatenation operator*} The concatenation operators & are predefined for every nonlimited, one-dimensional array type *T* with component type *C*. They have the following specifications:

4
```
function "&"(Left : T; Right : T) return T
function "&"(Left : T; Right : C) return T
function "&"(Left : C; Right : T) return T
function "&"(Left : C; Right : C) return T
```

Dynamic Semantics

5 {*evaluation (concatenation)* [partial]} For the evaluation of a concatenation with result type *T*, if both operands are of type *T*, the result of the concatenation is a one-dimensional array whose length is the sum of the lengths of its operands, and whose components comprise the components of the left operand followed by the components of the right operand. If the left operand is a null array, the result of the concatenation is the right operand. Otherwise, the lower bound of the result is determined as follows:

- 6 If the ultimate ancestor of the array type was defined by a constrained_array_definition, then the lower bound of the result is that of the index subtype;

  6.a **Reason:** This rule avoids Constraint_Error when using concatenation on an array type whose first subtype is constrained.

- 7 If the ultimate ancestor of the array type was defined by an unconstrained_array_definition, then the lower bound of the result is that of the left operand.

8 [The upper bound is determined by the lower bound and the length.] {*Index_Check* [partial]} {*check, language-defined (Index_Check)*} A check is made that the upper bound of the result of the concatenation belongs to the range of the index subtype, unless the result is a null array. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails.

9 If either operand is of the component type *C*, the result of the concatenation is given by the above rules, using in place of such an operand an array having this operand as its only component (converted to the component subtype) and having the lower bound of the index subtype of the array type as its lower bound. {*implicit subtype conversion (operand of concatenation)* [partial]}

9.a **Ramification:** The conversion might raise Constraint_Error. The conversion provides “sliding” for the component in the case of an array-of-arrays, consistent with the normal Ada 95 rules that allow sliding during parameter passing.

10 {*assignment operation (during evaluation of concatenation)*} The result of a concatenation is defined in terms of an assignment to an anonymous object, as for any function call (see 6.5).

10.a **Ramification:** This implies that value adjustment is performed as appropriate — see 7.6. We don't bother saying this for other predefined operators, even though they are all function calls, because this is the only one where it matters. It is the only one that can return a value having controlled parts.

NOTES

11 14 As for all predefined operators on modular types, the binary adding operators + and – on modular types include a final reduction modulo the modulus if the result is outside the base range of the type.

11.a **Implementation Note:** A full "modulus" operation need not be performed after addition or subtraction of modular types. For binary moduli, a simple mask is sufficient. For nonbinary moduli, a check after addition to see if the value is greater than the high bound of the base range can be followed by a conditional subtraction of the modulus. Conversely, a check after subtraction to see if a "borrow" was performed can be followed by a conditional addition of the modulus.

Examples

12 *Examples of expressions involving binary adding operators:*

13
```
Z + 0.1      -- Z has to be of a real type
```

14
```
"A" & "BCD"  -- concatenation of two string literals
'A' & "BCD"  -- concatenation of a character literal and a string literal
'A' & 'A'    -- concatenation of two character literals
```

Inconsistencies With Ada 83

14.a {*inconsistencies with Ada 83*} The lower bound of the result of concatenation, for a type whose first subtype is constrained, is now that of the index subtype. This is inconsistent with Ada 83, but generally only for Ada 83 programs that raise Constraint_Error. For example, the concatenation operator in

14.b
```
X : array(1..10) of Integer;
begin
X := X(6..10) & X(1..5);
```

14.c would raise Constraint_Error in Ada 83 (because the bounds of the result of the concatenation would be 6..15, which is outside of 1..10), but would succeed and swap the halves of X (as expected) in Ada 95.

Extensions to Ada 83

14.d {*extensions to Ada 83*} Concatenation is now useful for array types whose first subtype is constrained. When the result type of a concatenation is such an array type, Constraint_Error is avoided by effectively first sliding the left operand (if nonnull) so that its lower bound is that of the index subtype.

4.5.4 Unary Adding Operators

Static Semantics

1 {*unary adding operator*} {*operator (unary adding)*} {*+ operator*} {*operator (+)*} {*plus operator*} {*operator (plus)*} {*- operator*} {*operator (-)*} {*minus operator*} {*operator (minus)*} The unary adding operators + (identity) and – (negation) are predefined for every specific numeric type *T* with their conventional meaning. They have the following specifications:

2
```
function "+"(Right : T) return T
function "-"(Right : T) return T
```

NOTES

3 15 For modular integer types, the unary adding operator –, when given a nonzero operand, returns the result of subtracting the value of the operand from the modulus; for a zero operand, the result is zero.

4.5.5 Multiplying Operators

Static Semantics

1 {*multiplying operator*} {*operator (multiplying)*} {** operator*} {*operator (*)*} {*multiply operator*} {*operator (multiply)*} {*times operator*} {*operator (times)*} {*/ operator*} {*operator (/)*} {*divide operator*} {*operator (divide)*} {*mod operator*} {*operator (mod)*} {*rem operator*} {*operator (rem)*} The multiplying operators * (multiplication), / (division), **mod** (modulus), and **rem** (remainder) are predefined for every specific integer type *T*:

2
```
function "*"  (Left, Right : T) return T
function "/"  (Left, Right : T) return T
function "mod"(Left, Right : T) return T
function "rem"(Left, Right : T) return T
```

3 Signed integer multiplication has its conventional meaning.

Signed integer division and remainder are defined by the relation: 4

```
A = (A/B)*B + (A rem B)
```
5

where (A **rem** B) has the sign of A and an absolute value less than the absolute value of B. Signed integer division satisfies the identity: 6

```
(-A)/B = -(A/B) = A/(-B)
```
7

The signed integer modulus operator is defined such that the result of A **mod** B has the sign of B and an absolute value less than the absolute value of B; in addition, for some signed integer value N, this result satisfies the relation: 8

```
A = B*N + (A mod B)
```
9

The multiplying operators on modular types are defined in terms of the corresponding signed integer operators[, followed by a reduction modulo the modulus if the result is outside the base range of the type] [(which is only possible for the "*" operator)]. 10

> **Ramification:** The above identity satisfied by signed integer division is not satisfied by modular division because of the difference in effect of negation. 10.a

Multiplication and division operators are predefined for every specific floating point type *T*: 11

```
function "*"(Left, Right : T) return T
function "/"(Left, Right : T) return T
```
12

The following multiplication and division operators, with an operand of the predefined type Integer, are predefined for every specific fixed point type *T*: 13

```
function "*"(Left : T; Right : Integer) return T
function "*"(Left : Integer; Right : T) return T
function "/"(Left : T; Right : Integer) return T
```
14

[All of the above multiplying operators are usable with an operand of an appropriate universal numeric type.] The following additional multiplying operators for *root_real* are predefined[, and are usable when both operands are of an appropriate universal or root numeric type, and the result is allowed to be of type *root_real*, as in a number_declaration]: 15

> **Ramification:** These operators are analogous to the multiplying operators involving fixed or floating point types where *root_real* substitutes for the fixed or floating point type, and *root_integer* substitutes for Integer. Only values of the corresponding universal numeric types are implicitly convertible to these root numeric types, so these operators are really restricted to use with operands of a universal type, or the specified root numeric types. 15.a

```
function "*"(Left, Right : root_real) return root_real
function "/"(Left, Right : root_real) return root_real
```
16

```
function "*"(Left : root_real; Right : root_integer) return root_real
function "*"(Left : root_integer; Right : root_real) return root_real
function "/"(Left : root_real; Right : root_integer) return root_real
```
17

Multiplication and division between any two fixed point types are provided by the following two predefined operators: 18

> **Ramification:** *Universal_fixed* is the universal type for the class of fixed point types, meaning that these operators take operands of any fixed point types (not necessarily the same) and return a result that is implicitly (or explicitly) convertible to any fixed point type. 18.a

```
function "*"(Left, Right : universal_fixed) return universal_fixed
function "/"(Left, Right : universal_fixed) return universal_fixed
```
19

Name Resolution Rules

{*AI95-00364-01*} {*AI95-00420-01*} The above two fixed-fixed multiplying operators shall not be used in a context where the expected type for the result is itself *universal_fixed* [— the context has to identify some other numeric type to which the result is to be converted, either explicitly or implicitly]. Unless the predefined universal operator is identified using an expanded name with prefix denoting the package 19.1/2

Standard, an explicit conversion is required on the result when using the above fixed-fixed multiplication operator if either operand is of a type having a user-defined primitive multiplication operator such that:

19.2/2 • it is declared immediately within the same declaration list as the type; and

19.3/2 • both of its formal parameters are of a fixed-point type.

19.4/2 {*AI95-00364-01*} {*AI95-00420-01*} A corresponding requirement applies to the universal fixed-fixed division operator.

19.a/2 **Discussion:** The *small* of *universal_fixed* is infinitesimal; no loss of precision is permitted. However, fixed-fixed division is impractical to implement when an exact result is required, and multiplication will sometimes result in unanticipated overflows in such circumstances, so we require an explicit conversion to be inserted in expressions like A * B * C if A, B, and C are each of some fixed point type.

19.b/2 On the other hand, X := A * B; is permitted by this rule, even if X, A, and B are all of different fixed point types, since the expected type for the result of the multiplication is the type of X, which is necessarily not *universal_fixed*.

19.c/2 {*AI95-00364-01*} {*AI95-00420-01*} We have made these into Name Resolution rules to ensure that user-defined primitive fixed-fixed operators are not made unusable due to the presence of these universal fixed-fixed operators. But we do allow these operators to be used if prefixed by package Standard, so that they can be used in the definitions of user-defined operators.

Legality Rules

20/2 *This paragraph was deleted.*{*AI95-00364-01*}

Dynamic Semantics

21 The multiplication and division operators for real types have their conventional meaning. [For floating point types, the accuracy of the result is determined by the precision of the result type. For decimal fixed point types, the result is truncated toward zero if the mathematical result is between two multiples of the *small* of the specific result type (possibly determined by context); for ordinary fixed point types, if the mathematical result is between two multiples of the *small*, it is unspecified which of the two is the result. {*unspecified* [partial]}]

22 {*Division_Check* [partial]} {*check, language-defined (Division_Check)*} {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised by integer division, **rem**, and **mod** if the right operand is zero. [Similarly, for a real type *T* with *T'*Machine_Overflows True, division by zero raises Constraint_Error.]

NOTES

23 16 For positive A and B, A/B is the quotient and A **rem** B is the remainder when A is divided by B. The following relations are satisfied by the rem operator:

24
```
   A  rem (-B) =   A rem B
(-A) rem   B  = -(A rem B)
```

25 17 For any signed integer K, the following identity holds:

26
```
A mod B   =   (A + K*B) mod B
```

27 The relations between signed integer division, remainder, and modulus are illustrated by the following table:

28–29

A	B	A/B	A **rem** B	A **mod** B	A	B	A/B	A **rem** B	A **mod** B
10	5	2	0	0	-10	5	-2	0	0
11	5	2	1	1	-11	5	-2	-1	4
12	5	2	2	2	-12	5	-2	-2	3
13	5	2	3	3	-13	5	-2	-3	2
14	5	2	4	4	-14	5	-2	-4	1

30

A	B	A/B	A **rem** B	A **mod** B	A	B	A/B	A **rem** B	A **mod** B
10	-5	-2	0	0	-10	-5	2	0	0
11	-5	-2	1	-4	-11	-5	2	-1	-1
12	-5	-2	2	-3	-12	-5	2	-2	-2
13	-5	-2	3	-2	-13	-5	2	-3	-3
14	-5	-2	4	-1	-14	-5	2	-4	-4

Examples

31 *Examples of expressions involving multiplying operators:*

32
```
I : Integer := 1;
J : Integer := 2;
K : Integer := 3;
```

33
```
X : Real := 1.0;                      --  see 3.5.7
Y : Real := 2.0;
```

34
```
F : Fraction := 0.25;                 --  see 3.5.9
G : Fraction := 0.5;
```

35

Expression	*Value*	*Result Type*
I*J	2	*same as I and J, that is, Integer*
K/J	1	*same as K and J, that is, Integer*
K **mod** J	1	*same as K and J, that is, Integer*
X/Y	0.5	*same as X and Y, that is, Real*
F/2	0.125	*same as F, that is, Fraction*
3*F	0.75	*same as F, that is, Fraction*
0.75*G	0.375	*universal_fixed, implicitly convertible to any fixed point type*
Fraction(F*G)	0.125	*Fraction, as stated by the conversion*
Real(J)*Y	4.0	*Real, the type of both operands after conversion of J*

Incompatibilities With Ada 83

35.a.1/2 {*AI95-00364-01*} {*AI95-00420-01*} {*incompatibilities with Ada 83*} The universal fixed-fixed multiplying operators are now directly available (see below). Any attempt to use user-defined fixed-fixed multiplying operators will be ambiguous with the universal ones. The only way to use the user-defined operators is to fully qualify them in a prefix call. This problem was not documented during the design of Ada 95, and has been mitigated by Ada 2005.

Extensions to Ada 83

35.a {*extensions to Ada 83*} Explicit conversion of the result of multiplying or dividing two fixed point numbers is no longer required, provided the context uniquely determines some specific fixed point result type. This is to improve support for decimal fixed point, where requiring explicit conversion on every fixed-fixed multiply or divide was felt to be inappropriate.

35.b The type *universal_fixed* is covered by *universal_real*, so real literals and fixed point operands may be multiplied or divided directly, without any explicit conversions required.

Wording Changes from Ada 83

35.c We have used the normal syntax for function definition rather than a tabular format.

Incompatibilities With Ada 95

35.d/2 {*AI95-00364-01*} {*incompatibilities with Ada 95*} We have changed the resolution rules for the universal fixed-fixed multiplying operators to remove the incompatibility with Ada 83 discussed above. The solution is to hide the universal operators in some circumstances. As a result, some legal Ada 95 programs will require the insertion of an explicit conversion around a fixed-fixed multiply operator. This change is likely to catch as many bugs as it causes, since it is unlikely that the user wanted to use predefined operators when they had defined user-defined versions.

4.5.6 Highest Precedence Operators

Static Semantics

1 {*highest precedence operator*} {*operator (highest precedence)*} {*abs operator*} {*operator (abs)*} {*absolute value*} The highest precedence unary operator **abs** (absolute value) is predefined for every specific numeric type *T*, with the following specification:

2
```
function "abs"(Right : T) return T
```

3 {*not operator*} {*operator (not)*} {*logical operator: See also not operator*} The highest precedence unary operator **not** (logical negation) is predefined for every boolean type *T*, every modular type *T*, and for every one-dimensional array type *T* whose components are of a boolean type, with the following specification:

4
```
function "not"(Right : T) return T
```

5 The result of the operator **not** for a modular type is defined as the difference between the high bound of the base range of the type and the value of the operand. [For a binary modulus, this corresponds to a bit-wise complement of the binary representation of the value of the operand.]

6 The operator **not** that applies to a one-dimensional array of boolean components yields a one-dimensional boolean array with the same bounds; each component of the result is obtained by logical negation of the corresponding component of the operand (that is, the component that has the same index value). {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} {*Constraint_Error (raised by failure of run-time check)*} A check is made that each component of the result belongs to the component subtype; the exception Constraint_Error is raised if this check fails.

6.a **Discussion:** The check against the component subtype is per AI83-00535.

7 {*exponentiation operator*} {*operator (exponentiation)*} {*** operator*} {*operator (**)*} The highest precedence *exponentiation* operator ** is predefined for every specific integer type *T* with the following specification:

8
```
function "**"(Left : T; Right : Natural) return T
```

9 Exponentiation is also predefined for every specific floating point type as well as *root_real*, with the following specification (where *T* is *root_real* or the floating point type):

10
```
function "**"(Left : T; Right : Integer'Base) return T
```

11 {*exponent*} The right operand of an exponentiation is the *exponent*. The expression X**N with the value of the exponent N positive is equivalent to the expression X*X*...X (with N–1 multiplications) except that the multiplications are associated in an arbitrary order. With N equal to zero, the result is one. With the value of N negative [(only defined for a floating point operand)], the result is the reciprocal of the result using the absolute value of N as the exponent.

11.a **Ramification:** The language does not specify the order of association of the multiplications inherent in an exponentiation. For a floating point type, the accuracy of the result might depend on the particular association order chosen.

Implementation Permissions

{*Constraint_Error (raised by failure of run-time check)*} The implementation of exponentiation for the case of a negative exponent is allowed to raise Constraint_Error if the intermediate result of the repeated multiplications is outside the safe range of the type, even though the final result (after taking the reciprocal) would not be. (The best machine approximation to the final result in this case would generally be 0.0.) 12

NOTES
18 {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} As implied by the specification given above for exponentiation of an integer type, a check is made that the exponent is not negative. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails. 13

Inconsistencies With Ada 83

{*8652/0100*} {*AI95-00018-01*} {*inconsistencies with Ada 83*} The definition of "**" allows arbitrary association of the multiplications which make up the result. Ada 83 required left-to-right associations (confirmed by AI83-00137). Thus it is possible that "**" would provide a slightly different (and more potentially accurate) answer in Ada 95 than in the same Ada 83 program. 13.a.1/1

Wording Changes from Ada 83

We now show the specification for "**" for integer types with a parameter subtype of Natural rather than Integer for the exponent. This reflects the fact that Constraint_Error is raised if a negative value is provided for the exponent. 13.a

4.6 Type Conversions

[Explicit type conversions, both value conversions and view conversions, are allowed between closely related types as defined below. This clause also defines rules for value and view conversions to a particular subtype of a type, both explicit ones and those implicit in other constructs. {*subtype conversion: See type conversion*} {*type conversion*} {*conversion*} {*cast: See type conversion*}]{*subtype conversion: See also implicit subtype conversion*} {*type conversion, implicit: See implicit subtype conversion*} 1

Syntax

```
type_conversion ::=                                                    2
    subtype_mark(expression)
  | subtype_mark(name)
```

{*target subtype (of a type_conversion)*} The *target subtype* of a type_conversion is the subtype denoted by the subtype_mark. {*operand (of a type_conversion)*} The *operand* of a type_conversion is the expression or name within the parentheses; {*operand type (of a type_conversion)*} its type is the *operand type*. 3

{*convertible*} One type is *convertible* to a second type if a type_conversion with the first type as operand type and the second type as target type is legal according to the rules of this clause. Two types are convertible if each is convertible to the other. 4

Ramification: Note that "convertible" is defined in terms of legality of the conversion. Whether the conversion would raise an exception at run time is irrelevant to this definition. 4.a

{*8652/0017*} {*AI95-00184-01*} {*AI95-00330-01*} {*view conversion*} {*conversion (view)*} A type_conversion whose operand is the name of an object is called a *view conversion* if both its target type and operand type are tagged, or if it appears in a call as an actual parameter of mode **out** or **in out**; {*value conversion*} {*conversion (value)*} other type_conversions are called *value conversions*. {*super: See view conversion*} 5/2

Ramification: A view conversion to a tagged type can appear in any context that requires an object name, including in an object renaming, the prefix of a selected_component, and if the operand is a variable, on the left side of an assignment_statement. View conversions to other types only occur as actual parameters. Allowing view conversions of untagged types in all contexts seemed to incur an undue implementation burden. 5.a

5.b/2 {*AI95-00330-01*} A type conversion appearing as an **in out** parameter in a generic instantiation is not a view conversion; the second part of the rule only applies to subprogram calls, not instantiations.

Name Resolution Rules

6 {*expected type (type_conversion operand)* [partial]} The operand of a type_conversion is expected to be of any type.

6.a **Discussion:** This replaces the "must be determinable" wording of Ada 83. This is equivalent to (but hopefully more intuitive than) saying that the operand of a type_conversion is a “complete context.”

7 The operand of a view conversion is interpreted only as a name; the operand of a value conversion is interpreted as an expression.

7.a **Reason:** This formally resolves the syntactic ambiguity between the two forms of type_conversion, not that it really matters.

Legality Rules

8/2 {*AI95-00251-01*} In a view conversion for an untagged type, the target type shall be convertible (back) to the operand type.

8.a/2 **Reason:** Untagged view conversions appear only as [**in**] **out** parameters. Hence, the reverse conversion must be legal as well. The forward conversion must be legal even for an **out** parameter, because (for example) actual parameters of an access type are always copied in anyway.

Paragraphs 9 through 20 were reorganized and moved below.

8.b/2 **Discussion:** {*AI95-00251-01*} The entire Legality Rules section has been reorganized to eliminate an unintentional incompatibility with Ada 83. In rare cases, a type conversion between two types related by derivation is not allowed by Ada 95, while it is allowed in Ada 83. The reorganization fixes this. Much of the wording of the legality section is unchanged, but it is reordered and reformatted. Because of the limitations of our tools, we had to delete and replace nearly the entire section. The text of Ada 95 paragraphs 8 through 12, 14, 15, 17, 19, 20, and 24 are unchanged (just moved); these are now 24.1 through 24.5, 24.12, 24.13, 24.17, 24.19, 24.20, and 8.

21/2 {*AI95-00251-01*} {*type conversion (composite (non-array))*} {*conversion (composite (non-array))*} If there is a type that is an ancestor of both the target type and the operand type, or both types are class-wide types, then at least one of the following rules shall apply:

21.1/2 • {*AI95-00251-01*} {*type conversion (enumeration)*} {*conversion (enumeration)*} The target type shall be untagged; or

22 • The operand type shall be covered by or descended from the target type; or

22.a **Ramification:** This is a conversion toward the root, which is always safe.

23/2 • {*AI95-00251-01*} The operand type shall be a class-wide type that covers the target type; or

23.a **Ramification:** This is a conversion of a class-wide type toward the leaves, which requires a tag check. See Dynamic Semantics.

23.b/2 {*AI95-00251-01*} These two rules imply that a conversion from an ancestor type to a type extension is not permitted, as this would require specifying the values for additional components, in general, and changing the tag. An extension_aggregate has to be used instead, constructing a new value, rather than converting an existing value. However, a conversion from the class-wide type rooted at an ancestor type is permitted; such a conversion just verifies that the operand's tag is a descendant of the target.

23.1/2 • {*AI95-00251-01*} The operand and target types shall both be class-wide types and the specific type associated with at least one of them shall be an interface type.

23.c/2 **Ramification:** We allow converting any class-wide type T'Class to or from a class-wide interface type even if the specific type T does not have an appropriate interface ancestor, because some extension of T might have the needed ancestor. This is similar to a conversion of a class-wide type toward the leaves of the tree, and we need to be consistent. Of course, there is a run-time check that the actual object has the needed interface.

24/2 {*AI95-00251-01*} If there is no type that is the ancestor of both the target type and the operand type, and they are not both class-wide types, one of the following rules shall apply:

- {*AI95-00251-01*} {*type conversion (numeric)*} {*conversion (numeric)*} If the target type is a numeric type, then the operand type shall be a numeric type. 24.1/2
- {*AI95-00251-01*} {*type conversion (array)*} {*conversion (array)*} If the target type is an array type, then the operand type shall be an array type. Further: 24.2/2
  - {*AI95-00251-01*} The types shall have the same dimensionality; 24.3/2
  - {*AI95-00251-01*} Corresponding index types shall be convertible; {*convertible (required)* [partial]} 24.4/2
  - {*AI95-00251-01*} The component subtypes shall statically match; {*statically matching (required)* [partial]} 24.5/2
  - {*AI95-00392-01*} If the component types are anonymous access types, then the accessibility level of the operand type shall not be statically deeper than that of the target type; {*accessibility rule (type conversion, array components)* [partial]} 24.6/2

    Reason: For unrelated array types, the component types could have different accessibility, and we had better not allow a conversion of a local type into a global type, in case the local type points at local objects. We don't need a check for other types of components; such components necessarily are for related types, and either have the same accessibility or (for access discriminants) cannot be changed so the discriminant check will prevent problems. 24.b/2
  - {*AI95-00246-01*} Neither the target type nor the operand type shall be limited; 24.7/2

    Reason: We cannot allow conversions between unrelated limited types, as they may have different representations, and (since the types are limited), a copy cannot be made to reconcile the representations. 24.c/2
  - {*AI95-00251-01*} {*AI95-00363-01*} If the target type of a view conversion has aliased components, then so shall the operand type; and 24.8/2

    Reason: {*AI95-00363-01*} We cannot allow a view conversion from an object with unaliased components to an object with aliased components, because that would effectively allow pointers to unaliased components. This rule was missing from Ada 95. 24.d/2
  - {*AI95-00246-01*} {*AI95-00251-01*} The operand type of a view conversion shall not have a tagged, private, or volatile subcomponent. 24.9/2

    Reason: {*AI95-00246-01*} We cannot allow view conversions between unrelated might-be-by-reference types, as they may have different representations, and a copy cannot be made to reconcile the representations. 24.e/2

    Ramification: These rules only apply to unrelated array conversions; different (weaker) rules apply to conversions between related types. 24.f/2
- {*AI95-00230-01*} If the target type is *universal_access*, then the operand type shall be an access type. 24.10/2

  Discussion: Such a conversion cannot be written explicitly, of course, but it can be implicit (see below). 24.g/2
- {*AI95-00230-01*} {*AI95-00251-01*} {*type conversion (access)*} {*conversion (access)*} If the target type is a general access-to-object type, then the operand type shall be *universal_access* or an access-to-object type. Further, if the operand type is not *universal_access*: 24.11/2

  Discussion: The Legality Rules and Dynamic Semantics are worded so that a type_conversion T(X) (where T is an access type) is (almost) equivalent to the attribute_reference X.**all**'Access, where the result is of type T. The only difference is that the type_conversion accepts a null value, whereas the attribute_reference would raise Constraint_Error. 24.h/2
  - {*AI95-00251-01*} If the target type is an access-to-variable type, then the operand type shall be an access-to-variable type; 24.12/2

    Ramification: If the target type is an access-to-constant type, then the operand type can be access-to-constant or access-to-variable. 24.i/2
  - {*AI95-00251-01*} If the target designated type is tagged, then the operand designated type shall be convertible to the target designated type; {*convertible (required)* [partial]} 24.13/2
  - {*AI95-00251-01*} {*AI95-00363-01*} If the target designated type is not tagged, then the designated types shall be the same, and either: 24.14/2

24.15/2
- {*AI95-00363-01*} the designated subtypes shall statically match; or{*statically matching (required)* [partial]}

24.16/2
- {*AI95-00363-01*} {*AI95-00384-01*} the designated type shall be discriminated in its full view and unconstrained in any partial view, and one of the designated subtypes shall be unconstrained;

24.j/2 **Ramification:** {*AI95-00363-01*} This does not require that types have a partial view in order to allow the conversion, simply that any partial view that does exist is unconstrained.

24.k/2 {*AI95-00384-01*} This allows conversions both ways (either subtype can be unconstrained); while Ada 95 only allowed the conversion if the target subtype is unconstrained. We generally want type conversions to be symmetric; which type is the target shouldn't matter for legality.

24.l/2 **Reason:** {*AI95-00363-01*} If the visible partial view is constrained, we do not allow conversion between unconstrained and constrained subtypes. This means that whether the full type had discriminants is not visible to clients of the partial view.

24.m/2 **Reason:** These rules are designed to ensure that aliased array objects only *need* "dope" if their nominal subtype is unconstrained, but they can always *have* dope if required by the run-time model (since no sliding is permitted as part of access type conversion). By contrast, aliased discriminated objects will always *need* their discriminants stored with them, even if nominally constrained. (Here, we are assuming an implementation that represents an access value as a single pointer.)

24.17/2
- {*AI95-00251-01*} {*accessibility rule (type conversion)* [partial]} The accessibility level of the operand type shall not be statically deeper than that of the target type. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

24.n/2 **Ramification:** The access parameter case is handled by a run-time check. Run-time checks are also done in instance bodies.

24.18/2
- {*AI95-00230-01*} {*type conversion (access)*} {*conversion (access)*} If the target type is a pool-specific access-to-object type, then the operand type shall be *universal_access*.

24.o/2 **Reason:** This allows **null** to be converted to pool-specific types. Without it, **null** could be converted to general access types but not pool-specific ones, which would be too inconsistent. Remember that these rules only apply to unrelated types, so we don't have to talk about conversions to derived or other related types.

24.19/2
- {*AI95-00230-01*} {*AI95-00251-01*} {*type conversion (access)*} {*conversion (access)*} If the target type is an access-to-subprogram type, then the operand type shall be *universal_access* or an access-to-subprogram type. Further, if the operand type is not *universal_access*:

24.20/2
  - {*AI95-00251-01*} The designated profiles shall be subtype-conformant. {*subtype conformance (required)*}

24.21/2
  - {*AI95-00251-01*} {*accessibility rule (type conversion)* [partial]} The accessibility level of the operand type shall not be statically deeper than that of the target type. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. If the operand type is declared within a generic body, the target type shall be declared within the generic body.

24.p/2 **Reason:** The reason it is illegal to convert from an access-to-subprogram type declared in a generic body to one declared outside that body is that in an implementation that shares generic bodies, procedures declared inside the generic need to have a different calling convention — they need an extra parameter pointing to the data declared in the current instance. For procedures declared in the spec, that's OK, because the compiler can know about them at compile time of the instantiation.

Static Semantics

25 A type_conversion that is a value conversion denotes the value that is the result of converting the value of the operand to the target subtype.

26 A type_conversion that is a view conversion denotes a view of the object denoted by the operand. This view is a variable of the target type if the operand denotes a variable; otherwise it is a constant of the target type.

27 {*nominal subtype (associated with a type_conversion)* [partial]} The nominal subtype of a type_conversion is its target subtype.

Dynamic Semantics

28 {*evaluation (value conversion)* [partial]} {*corresponding value (of the target type of a conversion)*} {*conversion*} For the evaluation of a type_conversion that is a value conversion, the operand is evaluated, and then the value of the operand is *converted* to a *corresponding* value of the target type, if any. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} {*Constraint_Error (raised by failure of run-time check)*} If there is no value of the target type that corresponds to the operand value, Constraint_Error is raised[; this can only happen on conversion to a modular type, and only when the operand value is outside the base range of the modular type.] Additional rules follow:

- 29 {*type conversion (numeric)*} {*conversion (numeric)*} Numeric Type Conversion
  - 30 If the target and the operand types are both integer types, then the result is the value of the target type that corresponds to the same mathematical integer as the operand.
  - 31 If the target type is a decimal fixed point type, then the result is truncated (toward 0) if the value of the operand is not a multiple of the *small* of the target type.
  - 32 {*accuracy*} If the target type is some other real type, then the result is within the accuracy of the target type (see G.2, “Numeric Performance Requirements”, for implementations that support the Numerics Annex).

    32.a **Discussion:** An integer type might have more bits of precision than a real type, so on conversion (of a large integer), some precision might be lost.
  - 33 If the target type is an integer type and the operand type is real, the result is rounded to the nearest integer (away from zero if exactly halfway between two integers).

    33.a/2 **Discussion:** {*AI95-00267-01*} This was implementation defined in Ada 83. There seems no reason to preserve the nonportability in Ada 95. Round-away-from-zero is the conventional definition of rounding, and standard Fortran and COBOL both specify rounding away from zero, so for interoperability, it seems important to pick this. This is also the most easily “undone” by hand. Round-to-nearest-even is an alternative, but that is quite complicated if not supported by the hardware. In any case, this operation is not usually part of an inner loop, so predictability and portability are judged most important. A floating point attribute function Unbiased_Rounding is provided (see A.5.3) for those applications that require round-to-nearest-even, and a floating point attribute function Machine_Rounding (also see A.5.3) is provided for those applications that require the highest possible performance. “Deterministic” rounding is required for static conversions to integer as well. See 4.9.
- 34 {*type conversion (enumeration)*} {*conversion (enumeration)*} Enumeration Type Conversion
  - 35 The result is the value of the target type with the same position number as that of the operand value.
- 36 {*type conversion (array)*} {*conversion (array)*} Array Type Conversion
  - 37 {*Length_Check* [partial]} {*check, language-defined (Length_Check)*} If the target subtype is a constrained array subtype, then a check is made that the length of each dimension of the value of the operand equals the length of the corresponding dimension of the target subtype. The bounds of the result are those of the target subtype.
  - 38 {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} If the target subtype is an unconstrained array subtype, then the bounds of the result are obtained by converting each bound of the value of the operand to the corresponding index type of the target type. {*implicit subtype conversion (array bounds)* [partial]} For each nonnull index range, a check is made that the bounds of the range belong to the corresponding index subtype.

    38.a **Discussion:** Only nonnull index ranges are checked, per AI83-00313.
  - 39 In either array case, the value of each component of the result is that of the matching component of the operand value (see 4.5.2).

39.a **Ramification:** This applies whether or not the component is initialized.

39.1/2 • {*AI95-00392-01*} If the component types of the array types are anonymous access types, then a check is made that the accessibility level of the operand type is not deeper than that of the target type. {*Accessibility_Check* [partial]} {*check, language-defined (Accessibility_Check)*}

39.b/2 **Reason:** This check is needed for operands that are access parameters and in instance bodies. Other cases are handled by the legality rule given previously.

40 • {*type conversion (composite (non-array))*} {*conversion (composite (non-array))*} Composite (Non-Array) Type Conversion

41 • The value of each nondiscriminant component of the result is that of the matching component of the operand value.

41.a **Ramification:** This applies whether or not the component is initialized.

42 • [The tag of the result is that of the operand.] {*Tag_Check* [partial]} {*check, language-defined (Tag_Check)*} If the operand type is class-wide, a check is made that the tag of the operand identifies a (specific) type that is covered by or descended from the target type.

42.a **Ramification:** This check is certain to succeed if the operand type is itself covered by or descended from the target type.

42.b **Proof:** The fact that a type_conversion preserves the tag is stated officially in 3.9, "Tagged Types and Type Extensions"

43 • For each discriminant of the target type that corresponds to a discriminant of the operand type, its value is that of the corresponding discriminant of the operand value; {*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} if it corresponds to more than one discriminant of the operand type, a check is made that all these discriminants are equal in the operand value.

44 • For each discriminant of the target type that corresponds to a discriminant that is specified by the derived_type_definition for some ancestor of the operand type (or if class-wide, some ancestor of the specific type identified by the tag of the operand), its value in the result is that specified by the derived_type_definition.

44.a **Ramification:** It is a ramification of the rules for the discriminants of derived types that each discriminant of the result is covered either by this paragraph or the previous one. See 3.7.

45 • {*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} For each discriminant of the operand type that corresponds to a discriminant that is specified by the derived_type_definition for some ancestor of the target type, a check is made that in the operand value it equals the value specified for it.

46 • {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} For each discriminant of the result, a check is made that its value belongs to its subtype.

47 • {*type conversion (access)*} {*conversion (access)*} Access Type Conversion

48 • For an access-to-object type, a check is made that the accessibility level of the operand type is not deeper than that of the target type. {*Accessibility_Check* [partial]} {*check, language-defined (Accessibility_Check)*}

48.a **Ramification:** This check is needed for operands that are access parameters and in instance bodies.

48.b Note that this check can never fail for the implicit conversion to the anonymous type of an access parameter that is done when calling a subprogram with an access parameter.

49/2 • {*AI95-00230-01*} {*AI95-00231-01*} If the operand value is null, the result of the conversion is the null value of the target type.

49.a/2 **Ramification:** A conversion to an anonymous access type happens implicitly as part of initializing or assigning to an anonymous access object.

50 • If the operand value is not null, then the result designates the same object (or subprogram) as is designated by the operand value, but viewed as being of the target designated subtype (or

profile); any checks associated with evaluating a conversion to the target designated subtype are performed.

Ramification: The checks are certain to succeed if the target and operand designated subtypes statically match. 50.a

{*AI95-00231-01*} {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} {*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} {*Index_Check* [partial]} {*check, language-defined (Index_Check)*} {*Access_Check* [partial]} {*check, language-defined (Access_Check)*} After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null. 51/2

Ramification: {*AI95-00231-01*} The first check above is a Range_Check for scalar subtypes, a Discriminant_Check or Index_Check for access subtypes, and a Discriminant_Check for discriminated subtypes. The Length_Check for an array conversion is performed as part of the conversion to the target type. The check for exclusion of null is an Access_Check. 51.a/2

{*evaluation (view conversion)* [partial]} For the evaluation of a view conversion, the operand name is evaluated, and a new view of the object denoted by the operand is created, whose type is the target type; {*Length_Check* [partial]} {*check, language-defined (Length_Check)*} {*Tag_Check* [partial]} {*check, language-defined (Tag_Check)*} {*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} if the target type is composite, checks are performed as above for a value conversion. 52

The properties of this new view are as follows: 53

- {*8652/0017*} {*AI95-00184-01*} If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each nondiscriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the target subtype is indefinite, or if the operand type is a descendant of the target type and has discriminants that were not inherited from the target type; 54/1
- If the target type is tagged, then an assignment to the view assigns to the corresponding part of the object denoted by the operand; otherwise, an assignment to the view assigns to the object, after converting the assigned value to the subtype of the object (which might raise Constraint_Error); {*implicit subtype conversion (assignment to view conversion)* [partial]} 55
- Reading the value of the view yields the result of converting the value of the operand object to the target subtype (which might raise Constraint_Error), except if the object is of an access type and the view conversion is passed as an **out** parameter; in this latter case, the value of the operand object is used to initialize the formal parameter without checking against any constraint of the target subtype (see 6.4.1). {*implicit subtype conversion (reading a view conversion)* [partial]} 56

Reason: This ensures that even an **out** parameter of an access type is initialized reasonably. 56.a

{*Program_Error (raised by failure of run-time check)*} {*Constraint_Error (raised by failure of run-time check)*} If an Accessibility_Check fails, Program_Error is raised. Any other check associated with a conversion raises Constraint_Error if it fails. 57

Conversion to a type is the same as conversion to an unconstrained subtype of the type. 58

Reason: This definition is needed because the semantics of various constructs involves converting to a type, whereas an explicit type_conversion actually converts to a subtype. For example, the evaluation of a range is defined to convert the values of the expressions to the type of the range. 58.a

Ramification: A conversion to a scalar type, or, equivalently, to an unconstrained scalar subtype, can raise Constraint_Error if the value is outside the base range of the type. 58.b

NOTES

19 {*implicit subtype conversion* [distributed]} In addition to explicit type_conversions, type conversions are performed implicitly in situations where the expected type and the actual type of a construct differ, as is permitted by the type resolution rules (see 8.6). For example, an integer literal is of the type *universal_integer*, and is implicitly converted when assigned to a target of some specific integer type. Similarly, an actual parameter of a specific tagged type is implicitly converted when the corresponding formal parameter is of a class-wide type. 59

60 {*implicit subtype conversion* [distributed]} {*Constraint_Error (raised by failure of run-time check)*} Even when the expected and actual types are the same, implicit subtype conversions are performed to adjust the array bounds (if any) of an operand to match the desired target subtype, or to raise Constraint_Error if the (possibly adjusted) value does not satisfy the constraints of the target subtype.

61/2 20 {*AI95-00230-01*} A ramification of the overload resolution rules is that the operand of an (explicit) type_conversion cannot be an allocator, an aggregate, a string_literal, a character_literal, or an attribute_reference for an Access or Unchecked_Access attribute. Similarly, such an expression enclosed by parentheses is not allowed. A qualified_expression (see 4.7) can be used instead of such a type_conversion.

62 21 The constraint of the target subtype has no effect for a type_conversion of an elementary type passed as an **out** parameter. Hence, it is recommended that the first subtype be specified as the target to minimize confusion (a similar recommendation applies to renaming and generic formal **in out** objects).

Examples

63 *Examples of numeric type conversion:*

64
```
Real(2*J)       -- value is converted to floating point
Integer(1.6)    -- value is 2
Integer(-0.4)   -- value is 0
```

65 *Example of conversion between derived types:*

66
```
type A_Form is new B_Form;
```

67
```
X : A_Form;
Y : B_Form;
```

68
```
X := A_Form(Y);
Y := B_Form(X);  -- the reverse conversion
```

69 *Examples of conversions between array types:*

70
```
type Sequence is array (Integer range <>) of Integer;
subtype Dozen is Sequence(1 .. 12);
Ledger : array(1 .. 100) of Integer;
```

71
```
Sequence(Ledger)            -- bounds are those of Ledger
Sequence(Ledger(31 .. 42))  -- bounds are 31 and 42
Dozen(Ledger(31 .. 42))     -- bounds are those of Dozen
```

Incompatibilities With Ada 83

71.a {*incompatibilities with Ada 83*} A character_literal is not allowed as the operand of a type_conversion, since there are now two character types in package Standard.

71.b The component subtypes have to statically match in an array conversion, rather than being checked for matching constraints at run time.

71.c Because sliding of array bounds is now provided for operations where it was not in Ada 83, programs that used to raise Constraint_Error might now continue executing and produce a reasonable result. This is likely to fix more bugs than it creates.

Extensions to Ada 83

71.d {*extensions to Ada 83*} A type_conversion is considered the name of an object in certain circumstances (such a type_conversion is called a view conversion). In particular, as in Ada 83, a type_conversion can appear as an **in out** or **out** actual parameter. In addition, if the target type is tagged and the operand is the name of an object, then so is the type_conversion, and it can be used as the prefix to a selected_component, in an object_renaming_declaration, etc.

71.e We no longer require type-mark conformance between a parameter of the form of a type conversion, and the corresponding formal parameter. This had caused some problems for inherited subprograms (since there isn't really a type-mark for converted formals), as well as for renamings, formal subprograms, etc. See AI83-00245, AI83-00318, AI83-00547.

71.f We now specify “deterministic” rounding from real to integer types when the value of the operand is exactly between two integers (rounding is away from zero in this case).

71.g “Sliding” of array bounds (which is part of conversion to an array subtype) is performed in more cases in Ada 95 than in Ada 83. Sliding is not performed on the operand of a membership test, nor on the operand of a qualified_expression. It wouldn't make sense on a membership test, and we wish to retain a connection between subtype membership and subtype qualification. In general, a subtype membership test returns True if and only if a corresponding subtype qualification succeeds without raising an exception. Other operations that take arrays perform sliding.

Wording Changes from Ada 83

We no longer explicitly list the kinds of things that are not allowed as the operand of a type_conversion, except in a NOTE. 71.h

The rules in this clause subsume the rules for "parameters of the form of a type conversion," and have been generalized to cover the use of a type conversion as a name. 71.i

Incompatibilities With Ada 95

{*AI95-00246-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** Conversions between unrelated array types that are limited or (for view conversions) might be by-reference types are now illegal. The representations of two such arrays may differ, making the conversions impossible. We make the check here, because legality should not be based on representation properties. Such conversions are likely to be rare, anyway. There is a potential that this change would make a working program illegal (if the types have the same representation). 71.j/2

{*AI95-00363-01*} If a discriminated full type has a partial view (private type) that is constrained, we do not allow conversion between access-to-unconstrained and access-to-constrained subtypes designating the type. Ada 95 allowed this conversion and the declaration of various access subtypes, requiring that the designated object be constrained and thus making details of the implementation of the private type visible to the client of the private type. See 4.8 for more on this topic. 71.k/2

Extensions to Ada 95

{*AI95-00230-01*} {*extensions to Ada 95*} Conversion rules for *universal_access* were defined. These allow the use of anonymous access values in equality tests (see 4.5.2), and also allow the use of **null** in type conversions and other contexts that do not provide a single expected type. 71.l/2

{*AI95-00384-01*} A type conversion from an access-to-discriminated and unconstrained object to an access-to-discriminated and constrained one is allowed. Ada 95 only allowed the reverse conversion, which was weird and asymmetric. Of course, a constraint check will be performed for this conversion. 71.m/2

Wording Changes from Ada 95

{*8652/0017*} {*AI95-00184-01*} **Corrigendum:** Wording was added to ensure that view conversions are constrained, and that a tagged view conversion has a tagged object. Both rules are needed to avoid having a way to change the discriminants of a constrained object. 71.n/2

{*8652/0008*} {*AI95-00168-01*} **Corrigendum:** Wording was added to ensure that the aliased status of array components cannot change in a view conversion. This rule was needed to avoid having a way to change the discriminants of an aliased object. This rule was repealed later, as Ada 2005 allows changing the discriminants of an aliased object. 71.o/2

{*AI95-00231-01*} Wording was added to check subtypes that exclude null (see 3.10). 71.p/2

{*AI95-00251-01*} The organization of the legality rules was changed, both to make it clearer, and to eliminate an unintentional incompatibility with Ada 83. The old organization prevented type conversions between some types that were related by derivation (which Ada 83 always allowed). 71.q/2

{*AI95-00330-01*} Clarified that an untagged type conversion appearing as a generic actual parameter for a generic **in out** formal parameter is not a view conversion (and thus is illegal). This confirms the ACATS tests, so all implementations already follow this intepretation. 71.r/2

{*AI95-00363-01*} Rules added by the Corrigendum to eliminate problems with discriminants of aliased components changing were removed, as we now generally allow discriminants of aliased components to be changed. 71.s/2

{*AI95-00392-01*} Accessibility checks on conversions involving types with anonymous access components were added. These components have the level of the type, and conversions can be between types at different levels, which could cause dangling access values in the absence of such checks. 71.t/2

4.7 Qualified Expressions

[A qualified_expression is used to state explicitly the type, and to verify the subtype, of an operand that is either an expression or an aggregate. {*type conversion: See also qualified_expression*}] 1

Syntax

2 qualified_expression ::=
  subtype_mark'(expression) | subtype_mark'aggregate

Name Resolution Rules

3 {*operand (of a qualified_expression)* [partial]} The *operand* (the expression or aggregate) shall resolve to be of the type determined by the subtype_mark, or a universal type that covers it.

Dynamic Semantics

4 {*evaluation (qualified_expression)* [partial]} {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} {*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} {*Index_Check* [partial]} {*check, language-defined (Index_Check)*} The evaluation of a qualified_expression evaluates the operand (and if of a universal type, converts it to the type determined by the subtype_mark) and checks that its value belongs to the subtype denoted by the subtype_mark. {*implicit subtype conversion (qualified_expression)* [partial]} {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised if this check fails.

4.a **Ramification:** This is one of the few contexts in Ada 95 where implicit subtype conversion is not performed prior to a constraint check, and hence no "sliding" of array bounds is provided.

4.b **Reason:** Implicit subtype conversion is not provided because a qualified_expression with a constrained target subtype is essentially an assertion about the subtype of the operand, rather than a request for conversion. An explicit type_conversion can be used rather than a qualified_expression if subtype conversion is desired.

NOTES

5 22 When a given context does not uniquely identify an expected type, a qualified_expression can be used to do so. In particular, if an overloaded name or aggregate is passed to an overloaded subprogram, it might be necessary to qualify the operand to resolve its type.

Examples

6 *Examples of disambiguating expressions using qualification:*

7
```
type Mask is (Fix, Dec, Exp, Signif);
type Code is (Fix, Cla, Dec, Tnz, Sub);
```

8
```
Print (Mask'(Dec));  --  Dec is of type Mask
Print (Code'(Dec));  --  Dec is of type Code
```

9
```
for J in Code'(Fix) .. Code'(Dec) loop ... -- qualification needed for either Fix or Dec
for J in Code range Fix .. Dec loop ...     -- qualification unnecessary
for J in Code'(Fix) .. Dec loop ...         -- qualification unnecessary for Dec
```

10
```
Dozen'(1 | 3 | 5 | 7 => 2, others => 0) -- see 4.6
```

4.8 Allocators

1 [The evaluation of an allocator creates an object and yields an access value that designates the object. {*new: See allocator*} {*malloc: See allocator*} {*heap management: See also allocator*}]

Syntax

2 allocator ::=
  new subtype_indication | **new** qualified_expression

Name Resolution Rules

3/1 {*8652/0010*} {*AI95-00127-01*} {*expected type (allocator)* [partial]} The expected type for an allocator shall be a single access-to-object type with designated type *D* such that either *D* covers the type determined by the subtype_mark of the subtype_indication or qualified_expression, or the expected type is anonymous and the determined type is *D*'Class.

Discussion: See 8.6, “The Context of Overload Resolution” for the meaning of “shall be a single ... type whose ...” 3.a

Ramification: {*8652/0010*} {*AI95-00127-01*} An allocator is allowed as a controlling parameter of a dispatching call (see 3.9.2). 3.a.1/1

Legality Rules

{*initialized allocator*} An *initialized* allocator is an allocator with a qualified_expression. {*uninitialized allocator*} An *uninitialized* allocator is one with a subtype_indication. In the subtype_indication of an uninitialized allocator, a constraint is permitted only if the subtype_mark denotes an [unconstrained] composite subtype; if there is no constraint, then the subtype_mark shall denote a definite subtype. {*constructor: See initialized allocator*} 4

Ramification: For example, ... **new** S'Class ... (with no initialization expression) is illegal, but ... **new** S'Class'(X) ... is legal, and takes its tag and constraints from the initial value X. (Note that the former case cannot have a constraint.) 4.a

{*AI95-00287-01*} If the type of the allocator is an access-to-constant type, the allocator shall be an initialized allocator. 5/2

This paragraph was deleted.{*AI95-00287-01*} 5.a/2

{*AI95-00344-01*} If the designated type of the type of the allocator is class-wide, the accessibility level of the type determined by the subtype_indication or qualified_expression shall not be statically deeper than that of the type of the allocator. 5.1/2

Reason: This prevents the allocated object from outliving its type. 5.b/2

{*AI95-00416-01*} If the designated subtype of the type of the allocator has one or more unconstrained access discriminants, then the accessibility level of the anonymous access type of each access discriminant, as determined by the subtype_indication or qualified_expression of the allocator, shall not be statically deeper than that of the type of the allocator (see 3.10.2). 5.2/2

Reason: This prevents the allocated object from outliving its discriminants. 5.c/2

{*AI95-00366-01*} An allocator shall not be of an access type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. This rule does not apply in the body of a generic unit or within a body declared within the declarative region of a generic unit, if the type of the allocator is a descendant of a formal access type declared within the formal part of the generic unit. 5.3/2

Reason: An allocator for an access type that has Storage_Size specified to be zero is required to raise Storage_Error anyway. It's better to detect the error at compile-time, as the allocator might be executed infrequently. This also simplifies the rules for Pure units, where we do not want to allow any allocators for library-level access types, as they would represent state. 5.d/2

The last sentence covers the case of children of generics, and formal access types of formal packages of the generic unit. 5.e/2

Static Semantics

{*AI95-00363-01*} If the designated type of the type of the allocator is elementary, then the subtype of the created object is the designated subtype. If the designated type is composite, then the subtype of the created object is the designated subtype when the designated subtype is constrained or there is a partial view of the designated type that is constrained; otherwise, the created object is constrained by its initial value [(even if the designated subtype is unconstrained with defaults)]. {*constrained by its initial value* [partial]} 6/2

Discussion: See AI83-00331. 6.a

Reason: {*AI95-00363-01*} All objects created by an allocator are aliased, and most aliased composite objects need to be constrained so that access subtypes work reasonably. Problematic access subtypes are prohibited for types with a constrained partial view. 6.b/2

6.c/2 **Discussion:** {*AI95-00363-01*} If there is a constrained partial view of the type, this allows the objects to be unconstrained. This eliminates privacy breaking (we don't want the objects to act differently simply because they're allocated). Such a created object is effectively constrained by its initial value if the access type is an access-to-constant type, or the designated type is limited (in all views), but we don't need to state that here. It is implicit in other rules. Note, however, that a value of an access-to-constant type can designate a variable object via 'Access or conversion, and the variable object might be assigned by some other access path, and that assignment might alter the discriminants.

Dynamic Semantics

7/2 {*AI95-00373-01*} {*evaluation (allocator)* [partial]} For the evaluation of an initialized allocator, the evaluation of the qualified_expression is performed first. {*evaluation (initialized allocator)* [partial]} {*assignment operation (during evaluation of an initialized allocator)*} An object of the designated type is created and the value of the qualified_expression is converted to the designated subtype and assigned to the object. {*implicit subtype conversion (initialization expression of allocator)* [partial]}

7.a **Ramification:** The conversion might raise Constraint_Error.

8 {*evaluation (uninitialized allocator)* [partial]} For the evaluation of an uninitialized allocator, the elaboration of the subtype_indication is performed first. Then:

9/2
- {*AI95-00373-01*} {*assignment operation (during evaluation of an uninitialized allocator)*} If the designated type is elementary, an object of the designated subtype is created and any implicit initial value is assigned;

10/2
- {*8652/0002*} {*AI95-00171-01*} {*AI95-00373-01*} If the designated type is composite, an object of the designated type is created with tag, if any, determined by the subtype_mark of the subtype_indication. This object is then initialized by default (see 3.3.1) using the subtype_indication to determine its nominal subtype. {*Index_Check* [partial]} {*check, language-defined (Index_Check)*} {*Discriminant_Check* [partial]} {*check, language-defined (Discriminant_Check)*} A check is made that the value of the object belongs to the designated subtype. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails. This check and the initialization of the object are performed in an arbitrary order.

10.a **Discussion:** AI83-00150.

10.1/2 {*AI95-00344-01*} {*AI95-00416-01*} For any allocator, if the designated type of the type of the allocator is class-wide, then a check is made that the accessibility level of the type determined by the subtype_indication, or by the tag of the value of the qualified_expression, is not deeper than that of the type of the allocator. If the designated subtype of the allocator has one or more unconstrained access discriminants, then a check is made that the accessibility level of the anonymous access type of each access discriminant is not deeper than that of the type of the allocator. Program_Error is raised if either such check fails.{*Accessibility_Check* [partial]} {*check, language-defined (Accessibility_Check)*} {*Program_Error (raised by failure of run-time check)*}

10.b/2 **Reason:** {*AI95-00344-01*} The accessibility check on class-wide types prevents the allocated object from outliving its type. We need the run-time check in instance bodies, or when the type of the qualified_expression is class-wide (other cases are statically detected).

10.c/2 {*AI95-00416-01*} The accessibility check on access discriminants prevents the allocated object from outliving its discriminants.

10.2/2 {*AI95-00280-01*} If the object to be created by an allocator has a controlled or protected part, and the finalization of the collection of the type of the allocator (see 7.6.1) has started, Program_Error is raised.{*Allocation_Check* [partial]} {*check, language-defined (Allocation_Check)*} {*Program_Error (raised by failure of run-time check)*}

10.d/2 **Reason:** If the object has a controlled or protected part, its finalization is likely to be non-trivial. If the allocation was allowed, we could not know whether the finalization would actually be performed. That would be dangerous to otherwise safe abstractions, so we mandate a check here. On the other hand, if the finalization of the object will be trivial, we do not require (but allow) the check, as no real harm could come from late allocation.

Discussion: This check can only fail if an allocator is evaluated in code reached from a Finalize routine for a type declared in the same master. That's highly unlikely; Finalize routines are much more likely to be deallocating objects than allocating them. 10.e/2

{*AI95-00280-01*} If the object to be created by an allocator contains any tasks, and the master of the type of the allocator is completed, and all of the dependent tasks of the master are terminated (see 9.3), then Program_Error is raised.{*Allocation_Check* [partial]} {*check, language-defined (Allocation_Check)*} {*Program_Error (raised by failure of run-time check)*} 10.3/2

Reason: A task created after waiting for tasks has finished could depend on freed data structures, and certainly would never be awaited. 10.f/2

[If the created object contains any tasks, they are activated (see 9.2).] Finally, an access value that designates the created object is returned. 11

Bounded (Run-Time) Errors

{*AI95-00280-01*} {*bounded error (cause)* [partial]} It is a bounded error if the finalization of the collection of the type (see 7.6.1) of the allocator has started. If the error is detected, Program_Error is raised. Otherwise, the allocation proceeds normally. 11.1/2

Discussion: This check is required in some cases; see above. 11.a/2

NOTES

23 Allocators cannot create objects of an abstract type. See 3.9.3. 12

24 If any part of the created object is controlled, the initialization includes calls on corresponding Initialize or Adjust procedures. See 7.6. 13

25 As explained in 13.11, "Storage Management", the storage for an object allocated by an allocator comes from a storage pool (possibly user defined). {*Storage_Error (raised by failure of run-time check)*} The exception Storage_Error is raised by an allocator if there is not enough storage. Instances of Unchecked_Deallocation may be used to explicitly reclaim storage. 14

26 Implementations are permitted, but not required, to provide garbage collection (see 13.11.3). 15

Ramification: Note that in an allocator, the exception Constraint_Error can be raised by the evaluation of the qualified_expression, by the elaboration of the subtype_indication, or by the initialization. 15.a

Discussion: By default, the implementation provides the storage pool. The user may exercise more control over storage management by associating a user-defined pool with an access type. 15.b

Examples

Examples of allocators: 16

17
```
new Cell'(0, null, null)                          -- initialized explicitly, see 3.10.1
new Cell'(Value => 0, Succ => null, Pred => null) -- initialized explicitly
new Cell                                          -- not initialized
```

18
```
new Matrix(1 .. 10, 1 .. 20)                      -- the bounds only are given
new Matrix'(1 .. 10 => (1 .. 20 => 0.0))          -- initialized explicitly
```

19
```
new Buffer(100)                                   -- the discriminant only is given
new Buffer'(Size => 80, Pos => 0, Value => (1 .. 80 => 'A')) -- initialized explicitly
```

20
```
Expr_Ptr'(new Literal)                   -- allocator for access-to-class-wide type, see 3.9.1
Expr_Ptr'(new Literal'(Expression with 3.5))      -- initialized explicitly
```

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} The subtype_indication of an uninitialized allocator may not have an explicit constraint if the designated type is an access type. In Ada 83, this was permitted even though the constraint had no effect on the subtype of the created object. 20.a/1

Extensions to Ada 83

{*extensions to Ada 83*} Allocators creating objects of type *T* are now overloaded on access types designating *T*'Class and all class-wide types that cover *T*. 20.b

Implicit array subtype conversion (sliding) is now performed as part of an initialized allocator. 20.c

Wording Changes from Ada 83

20.d We have used a new organization, inspired by the ACID document, that makes it clearer what is the subtype of the created object, and what subtype conversions take place.

20.e Discussion of storage management issues, such as garbage collection and the raising of Storage_Error, has been moved to 13.11, “Storage Management”.

Inconsistencies With Ada 95

20.f/2 {*AI95-00363-01*} {*inconsistencies with Ada 95*} If the designated type has a constrained partial view, the allocated object can be unconstrained. This might cause the object to take up a different amount of memory, and might cause the operations to work where they previously would have raised Constraint_Error. It's unlikely that the latter would actually matter in a real program (Constraint_Error usually indicates a bug that would be fixed, not left in a program.) The former might cause Storage_Error to be raised at a different time than in an Ada 95 program.

Incompatibilities With Ada 95

20.g/2 {*AI95-00366-01*} {*incompatibilities with Ada 95*} An allocator for an access type that has Storage_Size specified to be zero is now illegal. Ada 95 allowed the allocator, but it had to raise Storage_Error if executed. The primary impact of this change should be to detect bugs.

Extensions to Ada 95

20.h/2 {*8652/0010*} {*AI95-00127-01*} {*extensions to Ada 95*} **Corrigendum:** An allocator can be a controlling parameter of a dispatching call. This was an oversight in Ada 95.

20.i/2 {*AI95-00287-01*} Initialized allocators are allowed when the designated type is limited.

Wording Changes from Ada 95

20.j/2 {*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Clarified the elaboration of per-object constraints for an uninitialized allocator.

20.k/2 {*AI95-00280-01*} Program_Error is now raised if the allocator occurs after the finalization of the collection or the waiting for tasks. This is not listed as an incompatibility as the Ada 95 behavior was unspecified, and Ada 95 implementations tend to generate programs that crash in this case.

20.l/2 {*AI95-00344-01*} Added accessibility checks to class-wide allocators. These checks could not fail in Ada 95 (as all of the designated types had to be declared at the same level, so the access type would necessarily have been at the same level or more nested than the type of allocated object).

20.m/2 {*AI95-00373-01*} Revised the description of evaluation of uninitialized allocators to use “initialized by default” so that the ordering requirements are the same for all kinds of objects that are default-initialized.

20.n/2 {*AI95-00416-01*} Added accessibility checks to access discriminants of allocators. These checks could not fail in Ada 95 as the discriminants always have the accessibility of the object.

4.9 Static Expressions and Static Subtypes

1 Certain expressions of a scalar or string type are defined to be static. Similarly, certain discrete ranges are defined to be static, and certain scalar and string subtypes are defined to be static subtypes. [{*static*} *Static* means determinable at compile time, using the declared properties or values of the program entities.] {*constant: See also static*}

1.a **Discussion:** As opposed to more elaborate data flow analysis, etc.

Language Design Principles

1.b For an expression to be static, it has to be calculable at compile time.

1.c Only scalar and string expressions are static.

1.d To be static, an expression cannot have any nonscalar, nonstring subexpressions (though it can have nonscalar constituent names). A static scalar expression cannot have any nonscalar subexpressions. There is one exception — a membership test for a string subtype can be static, and the result is scalar, even though a subexpression is nonscalar.

1.e The rules for evaluating static expressions are designed to maximize portability of static calculations.

2 {*static (expression)*} A static expression is [a scalar or string expression that is] one of the following:

- a numeric_literal; 3

  Ramification: A numeric_literal is always a static expression, even if its expected type is not that of a static subtype. However, if its value is explicitly converted to, or qualified by, a nonstatic subtype, the resulting expression is nonstatic. 3.a

- a string_literal of a static string subtype; 4

  Ramification: That is, the constrained subtype defined by the index range of the string is static. Note that elementary values don't generally have subtypes, while composite values do (since the bounds or discriminants are inherent in the value). 4.a

- a name that denotes the declaration of a named number or a static constant; 5

  Ramification: Note that enumeration literals are covered by the function_call case. 5.a

- a function_call whose *function*_name or *function*_prefix statically denotes a static function, and whose actual parameters, if any (whether given explicitly or by default), are all static expressions; 6

  Ramification: This includes uses of operators that are equivalent to function_calls. 6.a

- an attribute_reference that denotes a scalar value, and whose prefix denotes a static scalar subtype; 7

  Ramification: Note that this does not include the case of an attribute that is a function; a reference to such an attribute is not even an expression. See above for function *calls*. 7.a

  An implementation may define the staticness and other properties of implementation-defined attributes. 7.b

- an attribute_reference whose prefix statically denotes a statically constrained array object or array subtype, and whose attribute_designator is First, Last, or Length, with an optional dimension; 8

- a type_conversion whose subtype_mark denotes a static scalar subtype, and whose operand is a static expression; 9

- a qualified_expression whose subtype_mark denotes a static [(scalar or string)] subtype, and whose operand is a static expression; 10

  Ramification: This rules out the subtype_mark'aggregate case. 10.a

  Reason: Adding qualification to an expression shouldn't make it nonstatic, even for strings. 10.b

- a membership test whose simple_expression is a static expression, and whose range is a static range or whose subtype_mark denotes a static [(scalar or string)] subtype; 11

  Reason: Clearly, we should allow membership tests in exactly the same cases where we allow qualified_expressions. 11.a

- a short-circuit control form both of whose relations are static expressions; 12

- a static expression enclosed in parentheses. 13

  Discussion: {*static (value)*} Informally, we talk about a *static value*. When we do, we mean a value specified by a static expression. 13.a

  Ramification: The language requires a static expression in a number_declaration, a numeric type definition, a discrete_choice (sometimes), certain representation items, an attribute_designator, and when specifying the value of a discriminant governing a variant_part in a record_aggregate or extension_aggregate. 13.b

{*statically (denote)*} A name *statically denotes* an entity if it denotes the entity and: 14

- It is a direct_name, expanded name, or character_literal, and it denotes a declaration other than a renaming_declaration; or 15

- It is an attribute_reference whose prefix statically denotes some entity; or 16

- It denotes a renaming_declaration with a name that statically denotes the renamed entity. 17

  Ramification: Selected_components that are not expanded names and indexed_components do not statically denote things. 17.a

{*static (function)*} A *static function* is one of the following: 18

18.a **Ramification:** These are the functions whose calls can be static expressions.

19 • a predefined operator whose parameter and result types are all scalar types none of which are descendants of formal scalar types;

20 • a predefined concatenation operator whose result type is a string type;

21 • an enumeration literal;

22 • a language-defined attribute that is a function, if the prefix denotes a static scalar subtype, and if the parameter and result types are scalar.

23 In any case, a generic formal subprogram is not a static function.

24 {*static (constant)*} A *static constant* is a constant view declared by a full constant declaration or an object_renaming_declaration with a static nominal subtype, having a value defined by a static scalar expression or by a static string expression whose value has a length not exceeding the maximum length of a string_literal in the implementation.

24.a **Ramification:** A deferred constant is not static; the view introduced by the corresponding full constant declaration can be static.

24.b **Reason:** The reason for restricting the length of static string constants is so that compilers don't have to store giant strings in their symbol tables. Since most string constants will be initialized from string_literals, the length limit seems pretty natural. The reason for avoiding nonstring types is also to save symbol table space. We're trying to keep it cheap and simple (from the implementer's viewpoint), while still allowing, for example, the link name of a pragma Import to contain a concatenation.

24.c The length we're talking about is the maximum number of characters in the value represented by a string_literal, not the number of characters in the source representation; the quotes don't count.

25 {*static (range)*} A *static range* is a range whose bounds are static expressions, [or a range_attribute_reference that is equivalent to such a range.] {*static (discrete_range)*} A *static discrete_range* is one that is a static range or is a subtype_indication that defines a static scalar subtype. The base range of a scalar type is a static range, unless the type is a descendant of a formal scalar type.

26/2 {*AI95-00263-01*} {*static (subtype)*} A *static subtype* is either a *static scalar subtype* or a *static string subtype*. {*static (scalar subtype)*} A static scalar subtype is an unconstrained scalar subtype whose type is not a descendant of a formal type, or a constrained scalar subtype formed by imposing a compatible static constraint on a static scalar subtype. {*static (string subtype)*} A static string subtype is an unconstrained string subtype whose index subtype and component subtype are static, or a constrained string subtype formed by imposing a compatible static constraint on a static string subtype. In any case, the subtype of a generic formal object of mode **in out**, and the result subtype of a generic formal function, are not static.

26.a **Ramification:** String subtypes are the only composite subtypes that can be static.

26.b **Reason:** The part about generic formal objects of mode **in out** is necessary because the subtype of the formal is not required to have anything to do with the subtype of the actual. For example:

26.c
```
subtype Int10 is Integer range 1..10;
```

26.d
```
generic
    F : in out Int10;
procedure G;
```

26.e
```
procedure G is
begin
    case F is
        when 1..10 => null;
        -- Illegal!
    end case;
end G;
```

26.f
```
X : Integer range 1..20;
procedure I is new G(F => X); -- OK.
```

26.g The case_statement is illegal, because the subtype of F is not static, so the choices have to cover all values of Integer, not just those in the range 1..10. A similar issue arises for generic formal functions, now that function calls are object names.

27 {*static (constraint)*} The different kinds of *static constraint* are defined as follows:

- 28 A null constraint is always static;
- 29 {*static (range constraint)*} {*static (digits constraint)*} {*static (delta constraint)*} A scalar constraint is static if it has no range_constraint, or one with a static range;
- 30 {*static (index constraint)*} An index constraint is static if each discrete_range is static, and each index subtype of the corresponding array type is static;
- 31 {*static (discriminant constraint)*} A discriminant constraint is static if each expression of the constraint is static, and the subtype of each discriminant is static.

31.1/2 {*AI95-00311-01*} In any case, the constraint of the first subtype of a scalar formal type is neither static nor null.

32 {*statically (constrained)*} A subtype is *statically constrained* if it is constrained, and its constraint is static. An object is *statically constrained* if its nominal subtype is statically constrained, or if it is a static string constant.

Legality Rules

33 A static expression is evaluated at compile time except when it is part of the right operand of a static short-circuit control form whose value is determined by its left operand. This evaluation is performed exactly, without performing Overflow_Checks. For a static expression that is evaluated:

- 34 The expression is illegal if its evaluation fails a language-defined check other than Overflow_-Check.
- 35/2 {*AI95-00269-01*} If the expression is not part of a larger static expression and the expression is expected to be of a single specific type, then its value shall be within the base range of its expected type. Otherwise, the value may be arbitrarily large or small.

  35.a/2 **Ramification:** {*AI95-00269-01*} If the expression is expected to be of a universal type, or of “any integer type”, there are no limits on the value of the expression.

- 36/2 {*AI95-00269-01*} If the expression is of type *universal_real* and its expected type is a decimal fixed point type, then its value shall be a multiple of the *small* of the decimal type. This restriction does not apply if the expected type is a descendant of a formal scalar type (or a corresponding actual type in an instance).

  36.a **Ramification:** This means that a numeric_literal for a decimal type cannot have “extra” significant digits.

  36.b/2 **Reason:** {*AI95-00269-01*} The small is not known for a generic formal type, so we have to exclude formal types from this check.

37/2 {*AI95-00269-01*} {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), the above restrictions also apply in the private part of an instance of a generic unit.

37.a **Discussion:** Values outside the base range are not permitted when crossing from the “static” domain to the “dynamic” domain. This rule is designed to enhance portability of programs containing static expressions. Note that this rule applies to the exact value, not the value after any rounding or truncation. (See below for the rounding and truncation requirements.)

37.b Short-circuit control forms are a special case:

37.c
```
N: constant := 0.0;
X: constant Boolean := (N = 0.0) or else (1.0/N > 0.5); -- Static.
```

37.d The declaration of X is legal, since the divide-by-zero part of the expression is not evaluated. X is a static constant equal to True.

Implementation Requirements

38/2 {*AI95-00268-01*} {*AI95-00269-01*} For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the implementation shall round or truncate the value (according to the Machine_Rounds attribute of the expected type) to the nearest machine number of the expected type; if the value is exactly half-way between two machine numbers, the rounding performed is implementation-defined. If the expected type is a descendant of a formal type, or if the static expression appears in the body of an instance of a generic unit and the corresponding expression is nonstatic in the corresponding generic body, then no special rounding or truncating is required — normal accuracy rules apply (see Annex G).

38.a.1/2 **Implementation defined:** Rounding of real static expressions which are exactly half-way between two machine numbers.

38.a/2 **Reason:** {*AI95-00268-01*} Discarding extended precision enhances portability by ensuring that the value of a static constant of a real type is always a machine number of the type.

38.b When the expected type is a descendant of a formal floating point type, extended precision (beyond that of the machine numbers) can be retained when evaluating a static expression, to ease code sharing for generic instantiations. For similar reasons, normal (nondeterministic) rounding or truncating rules apply for descendants of a formal fixed point type.

38.b.1/2 {*AI95-00269-01*} There is no requirement for exact evaluation or special rounding in an instance body (unless the expression is static in the generic body). This eliminates a potential contract issue where the exact value of a static expression depends on the actual parameters (which could then affect the legality of other code).

38.c **Implementation Note:** Note that the implementation of static expressions has to keep track of plus and minus zero for a type whose Signed_Zeros attribute is True.

38.d/2 {*AI95-00100-01*} Note that the only machine numbers of a fixed point type are the multiples of the small, so a static conversion to a fixed-point type, or division by an integer, must do truncation to a multiple of small. It is not correct for the implementation to do all static calculations in infinite precision.

Implementation Advice

38.1/2 {*AI95-00268-01*} For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the rounding should be the same as the default rounding for the target system.

38.e/2 **Implementation Advice:** For a real static expression with a non-formal type that is not part of a larger static expression should be rounded the same as the target system.

NOTES

39 27 An expression can be static even if it occurs in a context where staticness is not required.

39.a **Ramification:** For example:

39.b
```
X : Float := Float'(1.0E+400) + 1.0 - Float'(1.0E+400);
```

39.c The expression is static, which means that the value of X must be exactly 1.0, independent of the accuracy or range of the run-time floating point implementation.

39.d The following kinds of expressions are never static: explicit_dereference, indexed_component, slice, **null**, aggregate, allocator.

40 28 A static (or run-time) type_conversion from a real type to an integer type performs rounding. If the operand value is exactly half-way between two integers, the rounding is performed away from zero.

40.a **Reason:** We specify this for portability. The reason for not choosing round-to-nearest-even, for example, is that this method is easier to undo.

40.b **Ramification:** The attribute Truncation (see A.5.3) can be used to perform a (static) truncation prior to conversion, to prevent rounding.

40.c **Implementation Note:** The value of the literal 0E99999999999999999999999999999999999999999999999999 is zero. The implementation must take care to evaluate such literals properly.

Examples

41 *Examples of static expressions:*

42
```
1 + 1       -- 2
abs(-10)*3  -- 30
```

43
```
Kilo : constant := 1000;
Mega : constant := Kilo*Kilo;   -- 1_000_000
Long : constant := Float'Digits*2;
```

44
```
Half_Pi    : constant := Pi/2;           -- see 3.3.2
Deg_To_Rad : constant := Half_Pi/90;
Rad_To_Deg : constant := 1.0/Deg_To_Rad; -- equivalent to 1.0/((3.14159_26536/2)/90)
```

Extensions to Ada 83

44.a {*extensions to Ada 83*} The rules for static expressions and static subtypes are generalized to allow more kinds of compile-time-known expressions to be used where compile-time-known values are required, as follows:

- 44.b Membership tests and short-circuit control forms may appear in a static expression.
- 44.c The bounds and length of statically constrained array objects or subtypes are static.
- 44.d The Range attribute of a statically constrained array subtype or object gives a static range.
- 44.e A type_conversion is static if the subtype_mark denotes a static scalar subtype and the operand is a static expression.
- 44.f All numeric literals are now static, even if the expected type is a formal scalar type. This is useful in case_statements and variant_parts, which both now allow a value of a formal scalar type to control the selection, to ease conversion of a package into a generic package. Similarly, named array aggregates are also permitted for array types with an index type that is a formal scalar type.

44.g The rules for the evaluation of static expressions are revised to require exact evaluation at compile time, and force a machine number result when crossing from the static realm to the dynamic realm, to enhance portability and predictability. Exact evaluation is not required for descendants of a formal scalar type, to simplify generic code sharing and to avoid generic contract model problems.

44.h Static expressions are legal even if an intermediate in the expression goes outside the base range of the type. Therefore, the following will succeed in Ada 95, whereas it might raise an exception in Ada 83:

44.i
```
type Short_Int is range -32_768 .. 32_767;
I : Short_Int := -32_768;
```

44.j This might raise an exception in Ada 83 because "32_768" is out of range, even though "–32_768" is not. In Ada 95, this will always succeed.

44.k Certain expressions involving string operations (in particular concatenation and membership tests) are considered static in Ada 95.

44.l The reason for this change is to simplify the rule requiring compile-time-known string expressions as the link name in an interfacing pragma, and to simplify the preelaborability rules.

Incompatibilities With Ada 83

44.m {*incompatibilities with Ada 83*} An Ada 83 program that uses an out-of-range static value is illegal in Ada 95, unless the expression is part of a larger static expression, or the expression is not evaluated due to being on the right-hand side of a short-circuit control form.

Wording Changes from Ada 83

44.n This clause (and 4.5.5, “Multiplying Operators”) subsumes the RM83 section on Universal Expressions.

44.o The existence of static string expressions necessitated changing the definition of static subtype to include string subtypes. Most occurrences of "static subtype" have been changed to "static scalar subtype", in order to preserve the effect of the Ada 83 rules. This has the added benefit of clarifying the difference between "static subtype" and "statically constrained subtype", which has been a source of confusion. In cases where we allow static string subtypes, we explicitly use phrases like "static string subtype" or "static (scalar or string) subtype", in order to clarify the meaning for those who have gotten used to the Ada 83 terminology.

44.p In Ada 83, an expression was considered nonstatic if it raised an exception. Thus, for example:

44.q
```
Bad: constant := 1/0; -- Illegal!
```

44.r was illegal because 1/0 was not static. In Ada 95, the above example is still illegal, but for a different reason: 1/0 is static, but there's a separate rule forbidding the exception raising.

Inconsistencies With Ada 95

44.s/2 {*AI95-00268-01*} {*inconsistencies with Ada 95*} **Amendment Correction:** Rounding of static real expressions is implementation-defined in Ada 2005, while it was specified as away from zero in (original) Ada 95. This could make subtle differences in programs. However, the original Ada 95 rule required rounding that (probably) differed from the target processor, thus creating anomalies where the value of a static expression was required to be different than the same expression evaluated at run-time.

Wording Changes from Ada 95

44.t/2 {*AI95-00263-01*} {*AI95-00268-01*} The Ada 95 wording that defined static subtypes unintentionally failed to exclude formal derived types that happen to be scalar (these aren't formal scalar types); and had a parenthetical remark excluding formal string types - but that was neither necessary nor parenthetical (it didn't follow from other wording). This issue also applies to the rounding rules for real static expressions.

44.u/2 {*AI95-00269-01*} Ada 95 didn't clearly define the bounds of a value of a static expression for universal types and for "any integer/float/fixed type". We also make it clear that we do not intend exact evaluation of static expressions in an instance body if the expressions aren't static in the generic body.

44.v/2 {*AI95-00311-01*} We clarify that the first subtype of a scalar formal type has a nonstatic, non-null constraint.

4.9.1 Statically Matching Constraints and Subtypes

Static Semantics

1/2 {*AI95-00311-01*} {*statically matching (for constraints)*} A constraint *statically matches* another constraint if:

1.1/2 • both are null constraints;

1.2/2 • both are static and have equal corresponding bounds or discriminant values;

1.3/2 • both are nonstatic and result from the same elaboration of a constraint of a subtype_indication or the same evaluation of a range of a discrete_subtype_definition; or

1.4/2 • {*AI95-00311-01*} both are nonstatic and come from the same formal_type_declaration.

2/2 {*AI95-00231-01*} {*AI95-00254-01*} {*statically matching (for subtypes)*} A subtype *statically matches* another subtype of the same type if they have statically matching constraints, and, for access subtypes, either both or neither exclude null. Two anonymous access-to-object subtypes statically match if their designated subtypes statically match, and either both or neither exclude null, and either both or neither are access-to-constant. Two anonymous access-to-subprogram subtypes statically match if their designated profiles are subtype conformant, and either both or neither exclude null.

2.a **Ramification:** Statically matching constraints and subtypes are the basis for subtype conformance of profiles (see 6.3.1).

2.b/2 **Reason:** Even though anonymous access types always represent different types, they can statically match. That's important so that they can be used widely. For instance, if this wasn't true, access parameters and access discriminants could never conform, so they couldn't be used in separate specifications.

3 {*statically matching (for ranges)*} Two ranges of the same type *statically match* if both result from the same evaluation of a range, or if both are static and have equal corresponding bounds.

3.a **Ramification:** The notion of static matching of ranges is used in 12.5.3, “Formal Array Types”; the index ranges of formal and actual constrained array subtypes have to statically match.

4 {*statically compatible (for a constraint and a scalar subtype)*} A constraint is *statically compatible* with a scalar subtype if it statically matches the constraint of the subtype, or if both are static and the constraint is compatible with the subtype. {*statically compatible (for a constraint and an access or composite subtype)*} A constraint is *statically compatible* with an access or composite subtype if it statically matches the constraint of the subtype, or if the subtype is unconstrained. {*statically compatible (for two subtypes)*} One subtype is *statically compatible* with a second subtype if the constraint of the first is statically compatible with the second subtype.

4.a **Discussion:** Static compatibility is required when constraining a parent subtype with a discriminant from a new discriminant_part. See 3.7. Static compatibility is also used in matching generic formal derived types.

4.b Note that statically compatible with a subtype does not imply compatible with a type. It is OK since the terms are used in different contexts.

Wording Changes from Ada 83

This subclause is new to Ada 95. 4.c

Wording Changes from Ada 95

{*AI95-00231-01*} {*AI95-00254-01*} Added static matching rules for null exclusions and anonymous access-to-subprogram types; both of these are new in Ada 2005. 4.d/2

{*AI95-00311-01*} We clarify that the constraint of the first subtype of a scalar formal type statically matches itself. 4.e/2

Section 5: Statements

[A statement defines an action to be performed upon its execution.] 1

{*AI95-00318-02*} [This section describes the general rules applicable to all statements. Some statements are discussed in later sections: Procedure_call_statements and return statements are described in 6, “Subprograms”. Entry_call_statements, requeue_statements, delay_statements, accept_statements, select_statements, and abort_statements are described in 9, “Tasks and Synchronization”. Raise_-statements are described in 11, “Exceptions”, and code_statements in 13. The remaining forms of statements are presented in this section.] 2/2

Wording Changes from Ada 83

{*AI95-00318-02*} The description of return statements has been moved to 6.5, “Return Statements”, so that it is closer to the description of subprograms. 2.a/2

5.1 Simple and Compound Statements - Sequences of Statements

[A statement is either simple or compound. A simple_statement encloses no other statement. A compound_statement can enclose simple_statements and other compound_statements.] 1

Syntax

```
sequence_of_statements ::= statement {statement}                    2

statement ::=                                                       3
  {label} simple_statement | {label} compound_statement

{AI95-00318-02} simple_statement ::= null_statement                 4/2
  | assignment_statement          | exit_statement
  | goto_statement                | procedure_call_statement
  | simple_return_statement       | entry_call_statement
  | requeue_statement             | delay_statement
  | abort_statement               | raise_statement
  | code_statement

{AI95-00318-02} compound_statement ::=                              5/2
    if_statement                  | case_statement
  | loop_statement                | block_statement
  | extended_return_statement
  | accept_statement              | select_statement

null_statement ::= null;                                            6

label ::= <<label_statement_identifier>>                            7

statement_identifier ::= direct_name                                8
```

The direct_name of a statement_identifier shall be an identifier (not an operator_symbol). 9

Name Resolution Rules

The direct_name of a statement_identifier shall resolve to denote its corresponding implicit declaration (see below). 10

Legality Rules

Distinct identifiers shall be used for all statement_identifiers that appear in the same body, including inner block_statements but excluding inner program units. 11

Static Semantics

12 For each statement_identifier, there is an implicit declaration (with the specified identifier) at the end of the declarative_part of the innermost block_statement or body that encloses the statement_identifier. The implicit declarations occur in the same order as the statement_identifiers occur in the source text. If a usage name denotes such an implicit declaration, the entity it denotes is the label, loop_statement, or block_statement with the given statement_identifier.

12.a **Reason:** We talk in terms of individual statement_identifiers here rather than in terms of the corresponding statements, since a given statement may have multiple statement_identifiers.

12.b A block_statement that has no explicit declarative_part has an implicit empty declarative_part, so this rule can safely refer to the declarative_part of a block_statement.

12.c The scope of a declaration starts at the place of the declaration itself (see 8.2). In the case of a label, loop, or block name, it follows from this rule that the scope of the implicit declaration starts before the first explicit occurrence of the corresponding name, since this occurrence is either in a statement label, a loop_statement, a block_statement, or a goto_statement. An implicit declaration in a block_statement may hide a declaration given in an outer program unit or block_statement (according to the usual rules of hiding explained in 8.3).

12.d The syntax rule for label uses statement_identifier which is a direct_name (not a defining_identifier), because labels are implicitly declared. The same applies to loop and block names. In other words, the label itself is not the defining occurrence; the implicit declaration is.

12.e We cannot consider the label to be a defining occurrence. An example that can tell the difference is this:

12.f
```
declare
    -- Label Foo is implicitly declared here.
begin
    for Foo in ... loop
        ...
        <<Foo>> -- Illegal.
        ...
    end loop;
end;
```

12.g The label in this example is hidden from itself by the loop parameter with the same name; the example is illegal. We considered creating a new syntactic category name, separate from direct_name and selector_name, for use in the case of statement labels. However, that would confuse the rules in Section 8, so we didn't do it.

Dynamic Semantics

13 {*execution (null_statement)* [partial]} The execution of a null_statement has no effect.

14/2 {*AI95-00318-02*} {*transfer of control*} A *transfer of control* is the run-time action of an exit_statement, return statement, goto_statement, or requeue_statement, selection of a terminate_alternative, raising of an exception, or an abort, which causes the next action performed to be one other than what would normally be expected from the other rules of the language. [As explained in 7.6.1, a transfer of control can cause the execution of constructs to be completed and then left, which may trigger finalization.]

15 {*execution (sequence_of_statements)* [partial]} The execution of a sequence_of_statements consists of the execution of the individual statements in succession until the sequence_ is completed.

15.a **Ramification:** It could be completed by reaching the end of it, or by a transfer of control.

NOTES

16 1 A statement_identifier that appears immediately within the declarative region of a named loop_statement or an accept_statement is nevertheless implicitly declared immediately within the declarative region of the innermost enclosing body or block_statement; in other words, the expanded name for a named statement is not affected by whether the statement occurs inside or outside a named loop or an accept_statement — only nesting within block_statements is relevant to the form of its expanded name.

Discussion: Each comment in the following example gives the expanded name associated with an entity declared in the task body: 16.a

16.b
```
task body Compute is
   Sum : Integer := 0;                       -- Compute.Sum
begin
 Outer:                                      -- Compute.Outer
   for I in 1..10 loop     -- Compute.Outer.I
    Blk:                                     -- Compute.Blk
      declare
         Sum : Integer := 0;                 -- Compute.Blk.Sum
      begin
         accept Ent(I : out Integer; J : in Integer) do
                                             -- Compute.Ent.I, Compute.Ent.J
            Compute.Ent.I := Compute.Outer.I;
          Inner:                             -- Compute.Blk.Inner
            for J in 1..10 loop
                                             -- Compute.Blk.Inner.J
               Sum := Sum + Compute.Blk.Inner.J * Compute.Ent.J;
            end loop Inner;
         end Ent;
         Compute.Sum := Compute.Sum + Compute.Blk.Sum;
      end Blk;
   end loop Outer;
   Record_Result(Sum);
end Compute;
```

Examples

Examples of labeled statements: 17

18
```
<<Here>> <<Ici>> <<Aqui>> <<Hier>> null;
```

19
```
<<After>> X := 1;
```

Extensions to Ada 83

{*extensions to Ada 83*} The requeue_statement is new. 19.a

Wording Changes from Ada 83

We define the syntactic category statement_identifier to simplify the description. It is used for labels, loop names, and block names. We define the entity associated with the implicit declarations of statement names. 19.b

Completion includes completion caused by a transfer of control, although RM83-5.1(6) did not take this view. 19.c

Extensions to Ada 95

{*AI95-00318-02*} {*extensions to Ada 95*} The extended_return_statement is new (simple_return_statement is merely renamed). 19.d/2

5.2 Assignment Statements

1 [An assignment_statement replaces the current value of a variable with the result of evaluating an expression.]

Syntax

2
```
assignment_statement ::=
  variable_name := expression;
```

3 The execution of an assignment_statement includes the evaluation of the expression and the *assignment* of the value of the expression into the *target*. {*assignment operation* [distributed]} {*assign: See assignment operation*} [An assignment operation (as opposed to an assignment_statement) is performed in other contexts as well, including object initialization and by-copy parameter passing.] {*target (of an assignment operation)*} {*target (of an assignment_statement)*} The *target* of an assignment operation is the view of the object to which a value is being assigned; the target of an assignment_statement is the variable denoted by the *variable*_name.

3.a **Discussion:** Don't confuse this notion of the “target” of an assignment with the notion of the “target object” of an entry call or requeue.

3.b Don't confuse the term “assignment operation” with the assignment_statement. The assignment operation is just one part of the execution of an assignment_statement. The assignment operation is also a part of the execution of various other constructs; see 7.6.1, “Completion and Finalization” for a complete list. Note that when we say, “such-and-such is assigned to so-and-so”, we mean that the assignment operation is being applied, and that so-and-so is the target of the assignment operation.

Name Resolution Rules

4/2 {*AI95-00287-01*} {*expected type (assignment_statement variable_name)* [partial]} The *variable_*name of an assignment_statement is expected to be of any type. {*expected type (assignment_statement expression)* [partial]} The expected type for the expression is the type of the target.

4.a **Implementation Note:** An assignment_statement as a whole is a "complete context," so if the *variable_*name of an assignment_statement is overloaded, the expression can be used to help disambiguate it. For example:

4.b
```
    type P1 is access R1;
    type P2 is access R2;
```

4.c
```
    function F return P1;
    function F return P2;
```

4.d
```
    X : R1;
  begin
    F.all := X;  -- Right hand side helps resolve left hand side
```

Legality Rules

5/2 {*AI95-00287-01*} The target [denoted by the *variable_*name] shall be a variable of a nonlimited type.

6 If the target is of a tagged class-wide type *T*'Class, then the expression shall either be dynamically tagged, or of type *T* and tag-indeterminate (see 3.9.2).

6.a **Reason:** This is consistent with the general rule that a single dispatching operation shall not have both dynamically tagged and statically tagged operands. Note that for an object initialization (as opposed to the assignment_statement), a statically tagged initialization expression is permitted, since there is no chance for confusion (or Tag_Check failure). Also, in an object initialization, tag-indeterminate expressions of any type covered by *T*'Class would be allowed, but with an assignment_statement, that might not work if the tag of the target was for a type that didn't have one of the dispatching operations in the tag-indeterminate expression.

Dynamic Semantics

7 {*execution (assignment_statement)* [partial]} For the execution of an assignment_statement, the *variable_*name and the expression are first evaluated in an arbitrary order.

7.a **Ramification:** Other rules of the language may require that the bounds of the variable be determined prior to evaluating the expression, but that does not necessarily require evaluation of the *variable_*name, as pointed out by the ACID.

8 When the type of the target is class-wide:

9
- {*controlling tag value (for the expression in an assignment_statement)* [partial]} If the expression is tag-indeterminate (see 3.9.2), then the controlling tag value for the expression is the tag of the target;

9.a **Ramification:** See 3.9.2, “Dispatching Operations of Tagged Types”.

10
- {*Tag_Check* [partial]} {*check, language-defined (Tag_Check)*} {*Constraint_Error (raised by failure of run-time check)*} Otherwise [(the expression is dynamically tagged)], a check is made that the tag of the value of the expression is the same as that of the target; if this check fails, Constraint_Error is raised.

11 The value of the expression is converted to the subtype of the target. [The conversion might raise an exception (see 4.6).] {*implicit subtype conversion (assignment_statement)* [partial]}

11.a **Ramification:** 4.6, “Type Conversions” defines what actions and checks are associated with subtype conversion. For non-array subtypes, it is just a constraint check presuming the types match. For array subtypes, it checks the lengths and slides if the target is constrained. “Sliding” means the array doesn't have to have the same bounds, so long as it is the same length.

In cases involving controlled types, the target is finalized, and an anonymous object might be used as an intermediate in the assignment, as described in 7.6.1, “Completion and Finalization”. {*assignment operation*} {*assignment operation (during execution of an assignment_statement)*} In any case, the converted value of the expression is then *assigned* to the target, which consists of the following two steps: 12

To be honest: 7.6.1 actually says that finalization happens always, but unless controlled types are involved, this finalization during an assignment_statement does nothing. 12.a

- The value of the target becomes the converted value. 13
- If any part of the target is controlled, its value is adjusted as explained in clause 7.6. {*adjustment (as part of assignment)* [partial]} 14

Ramification: If any parts of the object are controlled, abort is deferred during the assignment operation itself, but not during the rest of the execution of an assignment_statement. 14.a

NOTES
2 The tag of an object never changes; in particular, an assignment_statement does not change the tag of the target. 15

This paragraph was deleted.{*AI95-00363-01*} 16/2

Ramification: The implicit subtype conversion described above for assignment_statements is performed only for the value of the right-hand side expression as a whole; it is not performed for subcomponents of the value. 16.a

The determination of the type of the variable of an assignment_statement may require consideration of the expression if the variable name can be interpreted as the name of a variable designated by the access value returned by a function call, and similarly, as a component or slice of such a variable (see 8.6, “The Context of Overload Resolution”). 16.b

Examples

Examples of assignment statements: 17

18
```
Value := Max_Value - 1;
Shade := Blue;
```

19
```
Next_Frame(F)(M, N) := 2.5;        -- see 4.1.1
U := Dot_Product(V, W);            -- see 6.3
```

20
```
Writer := (Status => Open, Unit => Printer, Line_Count => 60);  -- see 3.8.1
Next_Car.all := (72074, null);    -- see 3.10.1
```

Examples involving scalar subtype conversions: 21

22
```
I, J : Integer range 1 .. 10 := 5;
K    : Integer range 1 .. 20 := 15;
 ...
```

23
```
I := J;  -- identical ranges
K := J;  -- compatible ranges
J := K;  -- will raise Constraint_Error if K > 10
```

Examples involving array subtype conversions: 24

25
```
A : String(1 .. 31);
B : String(3 .. 33);
 ...
```

26
```
A := B;  -- same number of components
```

27
```
A(1 .. 9)  := "tar sauce";
A(4 .. 12) := A(1 .. 9);  -- A(1 .. 12) = "tartar sauce"
```

NOTES
3 *Notes on the examples:* Assignment_statements are allowed even in the case of overlapping slices of the same array, because the *variable_*name and expression are both evaluated before copying the value into the variable. In the above example, an implementation yielding A(1 .. 12) = "tartartartar" would be incorrect. 28

Extensions to Ada 83

28.a {*extensions to Ada 83*} We now allow user-defined finalization and value adjustment actions as part of assignment_statements (see 7.6, "User-Defined Assignment and Finalization").

Wording Changes from Ada 83

28.b The special case of array assignment is subsumed by the concept of a subtype conversion, which is applied for all kinds of types, not just arrays. For arrays it provides "sliding". For numeric types it provides conversion of a value of a universal type to the specific type of the target. For other types, it generally has no run-time effect, other than a constraint check.

28.c We now cover in a general way in 3.7.2 the erroneous execution possible due to changing the value of a discriminant when the variable in an assignment_statement is a subcomponent that depends on discriminants.

Incompatibilities With Ada 95

28.d/2 {*AI95-00287-01*} {*incompatibilities with Ada 95*} The change of the limited check from a resolution rule to a legality rule is not quite upward compatible. For example.

28.e
```
type AccNonLim is access NonLim;
function Foo (Arg : in Integer) return AccNonLim;
type AccLim is access Lim;
function Foo (Arg : in Integer) return AccLim;
Foo(2).all := Foo(1).all;.
```

28.f where NonLim is a nonlimited type and Lim is a limited type. The assignment is legal in Ada 95 (only the first Foo would be considered), and is ambiguous in Ada 2005. We made the change because we want limited types to be as similar to nonlimited types as possible. Limited expressions are now allowed in all other contexts (with a similar incompatibility), and it would be odd if assignments had different resolution rules (which would eliminate ambiguities in some cases). Moreover, examples like this one are rare, as they depend on assigning into overloaded function calls.

5.3 If Statements

1 [An if_statement selects for execution at most one of the enclosed sequences_of_statements, depending on the (truth) value of one or more corresponding conditions.]

Syntax

2
```
if_statement ::=
   if condition then
     sequence_of_statements
  {elsif condition then
     sequence_of_statements}
  [else
     sequence_of_statements]
   end if;
```

3 condition ::= *boolean*_expression

Name Resolution Rules

4 {*expected type (condition)* [partial]} A condition is expected to be of any boolean type.

Dynamic Semantics

5 {*execution (if_statement)* [partial]} For the execution of an if_statement, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** True **then**), until one evaluates to True or all conditions are evaluated and yield False. If a condition evaluates to True, then the corresponding sequence_of_statements is executed; otherwise none of them is executed.

5.a **Ramification:** The part about all evaluating to False can't happen if there is an **else**, since that is herein considered equivalent to **elsif** True **then**.

Examples

Examples of if statements: 6

7
```
if Month = December and Day = 31 then
   Month := January;
   Day   := 1;
   Year  := Year + 1;
end if;
```

8
```
if Line_Too_Short then
   raise Layout_Error;
elsif Line_Full then
   New_Line;
   Put(Item);
else
   Put(Item);
end if;
```

9
```
if My_Car.Owner.Vehicle /= My_Car then            -- see 3.10.1
   Report ("Incorrect data");
end if;
```

5.4 Case Statements

1 [A case_statement selects for execution one of a number of alternative sequences_of_statements; the chosen alternative is defined by the value of an expression.]

Syntax

2
```
case_statement ::=
  case expression is
     case_statement_alternative
    {case_statement_alternative}
  end case;
```

3
```
case_statement_alternative ::=
  when discrete_choice_list =>
    sequence_of_statements
```

Name Resolution Rules

4 {*expected type (case expression)* [partial]} The expression is expected to be of any discrete type. {*expected type (case_statement_alternative discrete_choice)* [partial]} The expected type for each discrete_choice is the type of the expression.

Legality Rules

5 The expressions and discrete_ranges given as discrete_choices of a case_statement shall be static. [A discrete_choice **others**, if present, shall appear alone and in the last discrete_choice_list.]

6 The possible values of the expression shall be covered as follows:

- 7 If the expression is a name [(including a type_conversion or a function_call)] having a static and constrained nominal subtype, or is a qualified_expression whose subtype_mark denotes a static and constrained scalar subtype, then each non-**others** discrete_choice shall cover only values in that subtype, and each value of that subtype shall be covered by some discrete_choice [(either explicitly or by **others**)].

  7.a **Ramification:** Although not official names of objects, a value conversion still has a defined nominal subtype, namely its target subtype. See 4.6.

- 8 If the type of the expression is *root_integer*, *universal_integer*, or a descendant of a formal scalar type, then the case_statement shall have an **others** discrete_choice.

8.a **Reason:** This is because the base range is implementation defined for *root_integer* and *universal_integer*, and not known statically in the case of a formal scalar type.

9 • Otherwise, each value of the base range of the type of the expression shall be covered [(either explicitly or by **others**)].

10 Two distinct discrete_choices of a case_statement shall not cover the same value.

10.a **Ramification:** The goal of these coverage rules is that any possible value of the expression of a case_statement should be covered by exactly one discrete_choice of the case_statement, and that this should be checked at compile time. The goal is achieved in most cases, but there are two minor loopholes:

10.b • If the expression reads an object with an invalid representation (e.g. an uninitialized object), then the value can be outside the covered range. This can happen for static constrained subtypes, as well as nonstatic or unconstrained subtypes. It cannot, however, happen if the case_statement has the discrete_choice **others**, because **others** covers all values, even those outside the subtype.

10.c/2 • {*AI95-00114-01*} If the compiler chooses to represent the value of an expression of an unconstrained subtype in a way that includes values outside the bounds of the subtype, then those values can be outside the covered range. For example, if X: Integer := Integer'Last;, and the case expression is X+1, then the implementation might choose to produce the correct value, which is outside the bounds of Integer. (It might raise Constraint_Error instead.) This case can only happen for non-generic subtypes that are either unconstrained or nonstatic (or both). It can only happen if there is no **others** discrete_choice.

10.d In the uninitialized variable case, the value might be anything; hence, any alternative can be chosen, or Constraint_Error can be raised. (We intend to prevent, however, jumping to random memory locations and the like.) In the out-of-range case, the behavior is more sensible: if there is an **others**, then the implementation may choose to raise Constraint_Error on the evaluation of the expression (as usual), or it may choose to correctly evaluate the expression and therefore choose the **others** alternative. Otherwise (no **others**), Constraint_Error is raised either way — on the expression evaluation, or for the case_statement itself.

10.e For an enumeration type with a discontiguous set of internal codes (see 13.4), the only way to get values in between the proper values is via an object with an invalid representation; there is no “out-of-range” situation that can produce them.

Dynamic Semantics

11 {*execution (case_statement)* [partial]} For the execution of a case_statement the expression is first evaluated.

12 If the value of the expression is covered by the discrete_choice_list of some case_statement_-alternative, then the sequence_of_statements of the _alternative is executed.

13 {*Overflow_Check* [partial]} {*check, language-defined (Overflow_Check)*} {*Constraint_Error (raised by failure of run-time check)*} Otherwise (the value is not covered by any discrete_choice_list, perhaps due to being outside the base range), Constraint_Error is raised.

13.a **Ramification:** In this case, the value is outside the base range of its type, or is an invalid representation.

NOTES

14 4 The execution of a case_statement chooses one and only one alternative. Qualification of the expression of a case_statement by a static subtype can often be used to limit the number of choices that need be given explicitly.

Examples

15 *Examples of case statements:*

16
```
case Sensor is
   when Elevation => Record_Elevation(Sensor_Value);
   when Azimuth   => Record_Azimuth  (Sensor_Value);
   when Distance  => Record_Distance (Sensor_Value);
   when others    => null;
end case;
```

17
```
case Today is
   when Mon        => Compute_Initial_Balance;
   when Fri        => Compute_Closing_Balance;
   when Tue .. Thu => Generate_Report(Today);
   when Sat .. Sun => null;
end case;
```

18
```
case Bin_Number(Count) is
   when 1    => Update_Bin(1);
   when 2    => Update_Bin(2);
   when 3 | 4 =>
      Empty_Bin(1);
      Empty_Bin(2);
   when others   => raise Error;
end case;
```

Incompatibilities With Ada 83

18.a.1/1 {*incompatibilities with Ada 83*} In Ada 95, function_calls and type_conversions are names, whereas in Ada 83, they were expressions. Therefore, if the expression of a case_statement is a function_call or type_conversion, and the result subtype is static, it is illegal to specify a choice outside the bounds of the subtype. For this case in Ada 83 choices only are required to be in the base range of the type.

18.a.2/1 In addition, the rule about which choices must be covered is unchanged in Ada 95. Therefore, for a case_statement whose expression is a function_call or type_conversion, Ada 83 required covering all choices in the base range, while Ada 95 only requires covering choices in the bounds of the subtype. If the case_statement does not include an **others** discrete_choice, then a legal Ada 83 case_statement will be illegal in Ada 95 if the bounds of the subtype are different than the bounds of the base type.

Extensions to Ada 83

18.a {*extensions to Ada 83*} In Ada 83, the expression in a case_statement is not allowed to be of a generic formal type. This restriction is removed in Ada 95; an **others** discrete_choice is required instead.

18.b In Ada 95, a function call is the name of an object; this was not true in Ada 83 (see 4.1, “Names”). This change makes the following case_statement legal:

18.c
```
subtype S is Integer range 1..2;
function F return S;
case F is
   when 1 => ...;
   when 2 => ...;
   -- No others needed.
end case;
```

18.d Note that the result subtype given in a function renaming_declaration is ignored; for a case_statement whose expression calls a such a function, the full coverage rules are checked using the result subtype of the original function. Note that predefined operators such as "+" have an unconstrained result subtype (see 4.5.1). Note that generic formal functions do not have static result subtypes. Note that the result subtype of an inherited subprogram need not correspond to any namable subtype; there is still a perfectly good result subtype, though.

Wording Changes from Ada 83

18.e Ada 83 forgot to say what happens for “legally” out-of-bounds values.

18.f We take advantage of rules and terms (e.g. *cover a value*) defined for discrete_choices and discrete_choice_lists in 3.8.1, “Variant Parts and Discrete Choices”.

18.g In the Name Resolution Rule for the case expression, we no longer need RM83-5.4(3)'s “which must be determinable independently of the context in which the expression occurs, but using the fact that the expression must be of a discrete type,” because the expression is now a complete context. See 8.6, “The Context of Overload Resolution”.

18.h Since type_conversions are now defined as names, their coverage rule is now covered under the general rule for names, rather than being separated out along with qualified_expressions.

5.5 Loop Statements

1 [A loop_statement includes a sequence_of_statements that is to be executed repeatedly, zero or more times.]

Syntax

2 loop_statement ::=
  [*loop*_statement_identifier:]
    [iteration_scheme] **loop**
      sequence_of_statements
     end loop [*loop*_identifier];

3 iteration_scheme ::= **while** condition
  | **for** loop_parameter_specification

4 loop_parameter_specification ::=
  defining_identifier **in** [**reverse**] discrete_subtype_definition

5 If a loop_statement has a *loop*_statement_identifier, then the identifier shall be repeated after the **end loop**; otherwise, there shall not be an identifier after the **end loop**.

Static Semantics

6 {*loop parameter*} A loop_parameter_specification declares a *loop parameter*, which is an object whose subtype is that defined by the discrete_subtype_definition. {*parameter: See also loop parameter*}

Dynamic Semantics

7 {*execution (loop_statement)* [partial]} For the execution of a loop_statement, the sequence_of_statements is executed repeatedly, zero or more times, until the loop_statement is complete. The loop_statement is complete when a transfer of control occurs that transfers control out of the loop, or, in the case of an iteration_scheme, as specified below.

8 {*execution (loop_statement with a while iteration_scheme)* [partial]} For the execution of a loop_statement with a **while** iteration_scheme, the condition is evaluated before each execution of the sequence_of_statements; if the value of the condition is True, the sequence_of_statements is executed; if False, the execution of the loop_statement is complete.

9 {*execution (loop_statement with a for iteration_scheme)* [partial]} {*elaboration (loop_parameter_specification)* [partial]} For the execution of a loop_statement with a **for** iteration_scheme, the loop_parameter_specification is first elaborated. This elaboration creates the loop parameter and elaborates the discrete_subtype_definition. If the discrete_subtype_definition defines a subtype with a null range, the execution of the loop_statement is complete. Otherwise, the sequence_of_statements is executed once for each value of the discrete subtype defined by the discrete_subtype_definition (or until the loop is left as a consequence of a transfer of control). {*assignment operation (during execution of a for loop)*} Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order.

9.a **Ramification:** The order of creating the loop parameter and evaluating the discrete_subtype_definition doesn't matter, since the creation of the loop parameter has no side effects (other than possibly raising Storage_Error, but anything can do that).

NOTES

10 5 A loop parameter is a constant; it cannot be updated within the sequence_of_statements of the loop (see 3.3).

11 6 An object_declaration should not be given for a loop parameter, since the loop parameter is automatically declared by the loop_parameter_specification. The scope of a loop parameter extends from the loop_parameter_specification to the end of the loop_statement, and the visibility rules are such that a loop parameter is only visible within the sequence_of_statements of the loop.

11.a **Implementation Note:** An implementation could give a warning if a variable is hidden by a loop_parameter_specification.

12 7 The discrete_subtype_definition of a for loop is elaborated just once. Use of the reserved word **reverse** does not alter the discrete subtype defined, so that the following iteration_schemes are not equivalent; the first has a null range.

13
```
for J in reverse 1 .. 0
for J in 0 .. 1
```

13.a **Ramification:** If a loop_parameter_specification has a static discrete range, the subtype of the loop parameter is static.

Examples

14 *Example of a loop statement without an iteration scheme:*

15
```
loop
   Get(Current_Character);
   exit when Current_Character = '*';
end loop;
```

16 *Example of a loop statement with a* ***while*** *iteration scheme:*

17
```
while Bid(N).Price < Cut_Off.Price loop
   Record_Bid(Bid(N).Price);
   N := N + 1;
end loop;
```

18 *Example of a loop statement with a* ***for*** *iteration scheme:*

19
```
for J in Buffer'Range loop     --  works even with a null range
   if Buffer(J) /= Space then
      Put(Buffer(J));
   end if;
end loop;
```

20 *Example of a loop statement with a name:*

21
```
Summation:
   while Next /= Head loop       -- see 3.10.1
      Sum  := Sum + Next.Value;
      Next := Next.Succ;
   end loop Summation;
```

Wording Changes from Ada 83

21.a The constant-ness of loop parameters is specified in 3.3, “Objects and Named Numbers”.

5.6 Block Statements

1 [A block_statement encloses a handled_sequence_of_statements optionally preceded by a declarative_part.]

Syntax

2
```
block_statement ::=
  [block_statement_identifier:]
    [declare
       declarative_part]
     begin
       handled_sequence_of_statements
     end [block_identifier];
```

3 If a block_statement has a *block*_statement_identifier, then the identifier shall be repeated after the **end**; otherwise, there shall not be an identifier after the **end**.

Static Semantics

4 A block_statement that has no explicit declarative_part has an implicit empty declarative_part.

4.a **Ramification:** Thus, other rules can always refer to the declarative_part of a block_statement.

Dynamic Semantics

5 {*execution (block_statement)* [partial]} The execution of a block_statement consists of the elaboration of its declarative_part followed by the execution of its handled_sequence_of_statements.

Examples

6 *Example of a block statement with a local variable:*

7
```
Swap:
   declare
      Temp : Integer;
   begin
      Temp := V; V := U; U := Temp;
   end Swap;
```

7.a **Ramification:** If task objects are declared within a block_statement whose execution is completed, the block_statement is not left until all its dependent tasks are terminated (see 7.6). This rule applies to completion caused by a transfer of control.

7.b Within a block_statement, the block name can be used in expanded names denoting local entities such as Swap.Temp in the above example (see 4.1.3).

Wording Changes from Ada 83

7.c The syntax rule for block_statement now uses the syntactic category handled_sequence_of_statements.

5.7 Exit Statements

1 [An exit_statement is used to complete the execution of an enclosing loop_statement; the completion is conditional if the exit_statement includes a condition.]

Syntax

2 exit_statement ::=
  exit [*loop*_name] [**when** condition];

Name Resolution Rules

3 The *loop*_name, if any, in an exit_statement shall resolve to denote a loop_statement.

Legality Rules

4 {*apply (to a loop_statement by an exit_statement)*} Each exit_statement *applies to* a loop_statement; this is the loop_statement being exited. An exit_statement with a name is only allowed within the loop_-statement denoted by the name, and applies to that loop_statement. An exit_statement without a name is only allowed within a loop_statement, and applies to the innermost enclosing one. An exit_statement that applies to a given loop_statement shall not appear within a body or accept_statement, if this construct is itself enclosed by the given loop_statement.

Dynamic Semantics

5 {*execution (exit_statement)* [partial]} For the execution of an exit_statement, the condition, if present, is first evaluated. If the value of the condition is True, or if there is no condition, a transfer of control is done to complete the loop_statement. If the value of the condition is False, no transfer of control takes place.

NOTES

6 8 Several nested loops can be exited by an exit_statement that names the outer loop.

Examples

7 *Examples of loops with exit statements:*

8
```
for N in 1 .. Max_Num_Items loop
   Get_New_Item(New_Item);
   Merge_Item(New_Item, Storage_File);
   exit when New_Item = Terminal_Item;
end loop;
```

9
```
Main_Cycle:
   loop
      -- initial statements
      exit Main_Cycle when Found;
      -- final statements
   end loop Main_Cycle;
```

5.8 Goto Statements

1 [A goto_statement specifies an explicit transfer of control from this statement to a target statement with a given label.]

Syntax

2 goto_statement ::= **goto** *label*_name;

Name Resolution Rules

3 {*target statement (of a goto_statement)*} The *label*_name shall resolve to denote a label; the statement with that label is the *target statement*.

Legality Rules

4 The innermost sequence_of_statements that encloses the target statement shall also enclose the goto_statement. Furthermore, if a goto_statement is enclosed by an accept_statement or a body, then the target statement shall not be outside this enclosing construct.

4.a **Ramification:** The goto_statement can be a statement of an inner sequence_.

4.b It follows from the second rule that if the target statement is enclosed by such a construct, then the goto_statement cannot be outside.

Dynamic Semantics

5 {*execution (goto_statement)* [partial]} The execution of a goto_statement transfers control to the target statement, completing the execution of any compound_statement that encloses the goto_statement but does not enclose the target.

NOTES

6 9 The above rules allow transfer of control to a statement of an enclosing sequence_of_statements but not the reverse. Similarly, they prohibit transfers of control such as between alternatives of a case_statement, if_statement, or select_statement; between exception_handlers; or from an exception_handler of a handled_sequence_of_statements back to its sequence_of_statements.

Examples

7 *Example of a loop containing a goto statement:*

8
```
<<Sort>>
for I in 1 .. N-1 loop
   if A(I) > A(I+1) then
      Exchange(A(I), A(I+1));
      goto Sort;
   end if;
end loop;
```

Section 6: Subprograms

1 {*subprogram*} {*procedure*} {*function*} A subprogram is a program unit or intrinsic operation whose execution is invoked by a subprogram call. There are two forms of subprogram: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. The definition of a subprogram can be given in two parts: a subprogram declaration defining its interface, and a subprogram_body defining its execution. [Operators and enumeration literals are functions.]

1.a **To be honest:** A function call is an expression, but more specifically it is a name.

1.b/2 **Glossary entry:** {*Subprogram*} A subprogram is a section of a program that can be executed in various contexts. It is invoked by a subprogram call that may qualify the effect of the subprogram through the passing of parameters. There are two forms of subprograms: functions, which return values, and procedures, which do not.

1.c/2 **Glossary entry:** {*Function*} A function is a form of subprogram that returns a result and can be called as part of an expression.

1.d/2 **Glossary entry:** {*Procedure*} A procedure is a form of subprogram that does not return a result and can only be called by a statement.

2 {*callable entity*} A *callable entity* is a subprogram or entry (see Section 9). {*call*} A callable entity is invoked by a *call*; that is, a subprogram call or entry call. {*callable construct*} A *callable construct* is a construct that defines the action of a call upon a callable entity: a subprogram_body, entry_body, or accept_statement.

2.a **Ramification:** Note that “callable entity” includes predefined operators, enumeration literals, and abstract subprograms. “Call” includes calls of these things. They do not have callable constructs, since they don't have completions.

6.1 Subprogram Declarations

1 [A subprogram_declaration declares a procedure or function.]

Syntax

2/2 {*AI95-00218-03*} subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification;

3/2 *This paragraph was deleted.*{*AI95-00348-01*}

4/2 {*AI95-00348-01*} subprogram_specification ::=
  procedure_specification
 | function_specification

4.1/2 {*AI95-00348-01*} procedure_specification ::=
procedure defining_program_unit_name parameter_profile

4.2/2 {*AI95-00348-01*} function_specification ::=
function defining_designator parameter_and_result_profile

5 designator ::= [parent_unit_name .]identifier | operator_symbol

6 defining_designator ::= defining_program_unit_name | defining_operator_symbol

7 defining_program_unit_name ::= [parent_unit_name .]defining_identifier

8 [The optional parent_unit_name is only allowed for library units (see 10.1.1).]

9 operator_symbol ::= string_literal

10/2 {*AI95-00395-01*} The sequence of characters in an operator_symbol shall form a reserved word, a delimiter, or compound delimiter that corresponds to an operator belonging to one of the six categories of operators defined in clause 4.5.

10.a/2 **Reason:** {*AI95-00395-01*} The 'sequence of characters" of the string literal of the operator is a technical term (see 2.6), and does not include the surrounding quote characters. As defined in 2.2, lexical elements are "formed" from a sequence of characters. Spaces are not allowed, and upper and lower case is not significant. See 2.2 and 2.9 for rules related to the use of other_format characters in delimiters and reserved words.

11 defining_operator_symbol ::= operator_symbol

12 parameter_profile ::= [formal_part]

13/2 {*AI95-00231-01*} {*AI95-00318-02*} parameter_and_result_profile ::=
    [formal_part] **return** [null_exclusion] subtype_mark
  | [formal_part] **return** access_definition

14 formal_part ::=
   (parameter_specification {; parameter_specification})

15/2 {*AI95-00231-01*} parameter_specification ::=
    defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
  | defining_identifier_list : access_definition [:= default_expression]

16 mode ::= [**in**] | **in out** | **out**

Name Resolution Rules

17 {*formal parameter (of a subprogram)*} A *formal parameter* is an object [directly visible within a subprogram_body] that represents the actual parameter passed to the subprogram in a call; it is declared by a parameter_specification. {*expected type (parameter default_expression)* [partial]} For a formal parameter, the expected type for its default_expression, if any, is that of the formal parameter. {*parameter: See formal parameter*}

Legality Rules

18 {*parameter mode*} The *parameter mode* of a formal parameter conveys the direction of information transfer with the actual parameter: **in**, **in out**, or **out**. Mode **in** is the default, and is the mode of a parameter defined by an access_definition. The formal parameters of a function, if any, shall have the mode **in**.

18.a **Ramification:** Access parameters are permitted. This restriction to **in** parameters is primarily a methodological restriction, though it also simplifies implementation for some compiler technologies.

19 A default_expression is only allowed in a parameter_specification for a formal parameter of mode **in**.

20/2 {*AI95-00348-01*} {*requires a completion (subprogram_declaration)* [partial]} {*requires a completion (generic_subprogram_declaration)* [partial]} A subprogram_declaration or a generic_subprogram_declaration requires a completion: [a body, a renaming_declaration (see 8.5), or a pragma Import (see B.1)]. [A completion is not allowed for an abstract_subprogram_declaration (see 3.9.3) or a null_procedure_declaration (see 6.7).]

20.a/2 **Ramification:** {*AI95-00348-01*} Abstract subprograms and null procedures are not declared by subprogram_declarations, and so do not require completion. Protected subprograms are declared by subprogram_declarations, and so require completion. Note that an abstract subprogram is a subprogram, and a protected subprogram is a subprogram, but a generic subprogram is not a subprogram.

21 A name that denotes a formal parameter is not allowed within the formal_part in which it is declared, nor within the formal_part of a corresponding body or accept_statement.

21.a **Ramification:** By contrast, generic_formal_parameter_declarations are visible to subsequent declarations in the same generic_formal_part.

Static Semantics

22 {*profile*} The *profile* of (a view of) a callable entity is either a parameter_profile or parameter_and_result_profile[; it embodies information about the interface to that entity — for example,

the profile includes information about parameters passed to the callable entity. All callable entities have a profile — enumeration literals, other subprograms, and entries. An access-to-subprogram type has a designated profile.] Associated with a profile is a calling convention. A subprogram_declaration declares a procedure or a function, as indicated by the initial reserved word, with name and profile as given by its specification.

23/2 {*AI95-00231-01*} {*AI95-00318-02*} {*nominal subtype (of a formal parameter)* [partial]} The nominal subtype of a formal parameter is the subtype determined by the optional null_exclusion and the subtype_mark, or defined by the access_definition, in the parameter_specification. The nominal subtype of a function result is the subtype determined by the optional null_exclusion and the subtype_mark, or defined by the access_definition, in the parameter_and_result_profile. {*nominal subtype (of a function result)* [partial]}

24/2 {*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00318-02*} {*access parameter*} An *access parameter* is a formal **in** parameter specified by an access_definition. {*access result type*} An *access result type* is a function result type specified by an access_definition. An access parameter or result type is of an anonymous access type (see 3.10). [Access parameters of an access-to-object type allow dispatching calls to be controlled by access values. Access parameters of an access-to-subprogram type permit calls to subprograms passed as parameters irrespective of their accessibility level.]

24.a/2 **Discussion:** {*AI95-00318-02*} Access result types have normal accessibility and thus don't have any special properties worth noting here.

25 {*subtypes (of a profile)*} The *subtypes of a profile* are:

- 26 For any non-access parameters, the nominal subtype of the parameter.
- 27/2 {*AI95-00254-01*} For any access parameters of an access-to-object type, the designated subtype of the parameter type.
- 27.1/2 {*AI95-00254-01*} For any access parameters of an access-to-subprogram type, the subtypes of the profile of the parameter type.
- 28/2 {*AI95-00231-01*} {*AI95-00318-02*} For any non-access result, the nominal subtype of the function result.
- 28.1/2 {*AI95-00318-02*} For any access result type of an access-to-object type, the designated subtype of the result type.
- 28.2/2 {*AI95-00318-02*} For any access result type of an access-to-subprogram type, the subtypes of the profile of the result type.

29 [{*types (of a profile)*} The *types of a profile* are the types of those subtypes.]

30/2 {*AI95-00348-01*} [A subprogram declared by an abstract_subprogram_declaration is abstract; a subprogram declared by a subprogram_declaration is not. See 3.9.3, “Abstract Types and Subprograms”. Similarly, a procedure defined by a null_procedure_declaration is a null procedure; a procedure declared by a subprogram_declaration is not. See 6.7, “Null Procedures”.]

30.1/2 {*AI95-00218-03*} [An overriding_indicator is used to indicate whether overriding is intended. See 8.3.1, “Overriding Indicators”.]

Dynamic Semantics

31/2 {*AI95-00348-01*} {*elaboration (subprogram_declaration)* [partial]} The elaboration of a subprogram_declaration has no effect.

NOTES

32 1 A parameter_specification with several identifiers is equivalent to a sequence of single parameter_specifications, as explained in 3.3.

33 2 Abstract subprograms do not have bodies, and cannot be used in a nondispatching call (see 3.9.3, “Abstract Types and Subprograms”).

34 3 The evaluation of default_expressions is caused by certain calls, as described in 6.4.1. They are not evaluated during the elaboration of the subprogram declaration.

35 4 Subprograms can be called recursively and can be called concurrently from multiple tasks.

Examples

36 *Examples of subprogram declarations:*

37
```
procedure Traverse_Tree;
procedure Increment(X : in out Integer);
procedure Right_Indent(Margin : out Line_Size);          -- see 3.5.4
procedure Switch(From, To : in out Link);                -- see 3.10.1
```

38
```
function Random return Probability;                      -- see 3.5.7
```

39
```
function Min_Cell(X : Link) return Cell;                 -- see 3.10.1
function Next_Frame(K : Positive) return Frame;          -- see 3.10
function Dot_Product(Left, Right : Vector) return Real;  -- see 3.6
```

40
```
function "*"(Left, Right : Matrix) return Matrix;        -- see 3.6
```

41 *Examples of **in** parameters with default expressions:*

42
```
procedure Print_Header(Pages  : in Natural;
            Header : in Line    :=  (1 .. Line'Last => ' ');  -- see 3.6
            Center : in Boolean := True);
```

Extensions to Ada 83

42.a {*extensions to Ada 83*} The syntax for abstract_subprogram_declaration is added. The syntax for parameter_specification is revised to allow for access parameters (see 3.10)

42.b Program units that are library units may have a parent_unit_name to indicate the parent of a child (see Section 10).

Wording Changes from Ada 83

42.c We have incorporated the rules from RM83-6.5, “Function Subprograms” here and in 6.3, “Subprogram Bodies”

42.d We have incorporated the definitions of RM83-6.6, “Parameter and Result Type Profile - Overloading of Subprograms” here.

42.e The syntax rule for defining_operator_symbol is new. It is used for the defining occurrence of an operator_symbol, analogously to defining_identifier. Usage occurrences use the direct_name or selector_name syntactic categories. The syntax rules for defining_designator and defining_program_unit_name are new.

Extensions to Ada 95

42.f/2 {*AI95-00218-03*} {*extensions to Ada 95*} Subprograms now allow overriding_indicators for better error checking of overriding.

42.g/2 {*AI95-00231-01*} An optional null_exclusion can be used in a formal parameter declaration. Similarly, an optional null_exclusion can be used in a function result.

42.h/2 {*AI95-00318-02*} The return type of a function can be an anonymous access type.

Wording Changes from Ada 95

42.i/2 {*AI95-00254-01*} A description of the purpose of anonymous access-to-subprogram parameters and the definition of the profile of subprograms containing them was added.

42.j/2 {*AI95-00348-01*} Split the production for subprogram_specification in order to make the declaration of null procedures (see 6.7) easier.

42.k/2 {*AI95-00348-01*} Moved the Syntax and Dynamic Semantics for abstract_subprogram_declaration to 3.9.3, so that the syntax and semantics are together. This also keeps abstract and null subprograms similar.

42.l/2 {*AI95-00395-01*} Revised to allow other_format characters in operator_symbols in the same way as the underlying constructs.

6.2 Formal Parameter Modes

1 [A parameter_specification declares a formal parameter of mode **in**, **in out**, or **out**.]

Static Semantics

2 {*pass by copy*} {*by copy parameter passing*} {*copy parameter passing*} {*pass by reference*} {*by reference parameter passing*} {*reference parameter passing*} A parameter is passed either *by copy* or *by reference*. [When a parameter is passed by copy, the formal parameter denotes a separate object from the actual parameter, and any information transfer between the two occurs only before and after executing the subprogram. When a parameter is passed by reference, the formal parameter denotes (a view of) the object denoted by the actual parameter; reads and updates of the formal parameter directly reference the actual parameter object.]

3 {*by-copy type*} A type is a *by-copy type* if it is an elementary type, or if it is a descendant of a private type whose full type is a by-copy type. A parameter of a by-copy type is passed by copy.

4 {*by-reference type*} A type is a *by-reference type* if it is a descendant of one of the following:

- 5 a tagged type;
- 6 a task or protected type;
- 7 a nonprivate type with the reserved word **limited** in its declaration;

  7.a **Ramification:** A limited private type is by-reference only if it falls under one of the other categories.

- 8 a composite type with a subcomponent of a by-reference type;
- 9 a private type whose full type is a by-reference type.

10 A parameter of a by-reference type is passed by reference. {*associated object (of a value of a by-reference type)*} Each value of a by-reference type has an associated object. For a parenthesized expression, qualified_expression, or type_conversion, this object is the one associated with the operand.

10.a **Ramification:** By-reference parameter passing makes sense only if there is an object to reference; hence, we define such an object for each case.

10.b Since tagged types are by-reference types, this implies that every value of a tagged type has an associated object. This simplifies things, because we can define the tag to be a property of the object, and not of the value of the object, which makes it clearer that object tags never change.

10.c We considered simplifying things even more by making every value (and therefore every expression) have an associated object. After all, there is little semantic difference between a constant object and a value. However, this would cause problems for untagged types. In particular, we would have to do a constraint check on every read of a type conversion (or a renaming thereof) in certain cases.

10.d/2 {*AI95-00318-02*} We do not want this definition to depend on the view of the type; privateness is essentially ignored for this definition. Otherwise, things would be confusing (does the rule apply at the call site, at the site of the declaration of the subprogram, at the site of the return statement?), and requiring different calls to use different mechanisms would be an implementation burden.

10.e C.6, “Shared Variable Control” says that a composite type with an atomic or volatile subcomponent is a by-reference type, among other things.

10.f {*associated object (of a value of a limited type)*} Every value of a limited by-reference type is the value of one and only one limited object. The *associated object* of a value of a limited by-reference type is the object whose value it represents. {*same value (for a limited type)*} Two values of a limited by-reference type are the *same* if and only if they represent the value of the same object.

10.g We say “by-reference” above because these statements are not always true for limited private types whose underlying type is nonlimited (unfortunately).

11 {*unspecified* [partial]} For parameters of other types, it is unspecified whether the parameter is passed by copy or by reference.

11.a **Discussion:** There is no need to incorporate the discussion of AI83-00178, which requires pass-by-copy for certain kinds of actual parameters, while allowing pass-by-reference for others. This is because we explicitly indicate that a function creates an anonymous constant object for its result, unless the type is a return-by-reference type (see 6.5). We also provide a special dispensation for instances of Unchecked_Conversion to return by reference, even if the result type is not a return-by-reference type (see 13.9).

Bounded (Run-Time) Errors

12 {*distinct access paths*} {*access paths (distinct)*} {*aliasing: See distinct access paths*} {*bounded error (cause)* [partial]} If one name denotes a part of a formal parameter, and a second name denotes a part of a distinct formal parameter or an object that is not part of a formal parameter, then the two names are considered *distinct access paths*. If an object is of a type for which the parameter passing mechanism is not specified, then it is a bounded error to assign to the object via one access path, and then read the value of the object via a distinct access path, unless the first access path denotes a part of a formal parameter that no longer exists at the point of the second access [(due to leaving the corresponding callable construct).] {*Program_Error (raised by failure of run-time check)*} The possible consequences are that Program_Error is raised, or the newly assigned value is read, or some old value of the object is read.

12.a **Discussion:** For example, if we call "P(X => Global_Variable, Y => Global_Variable)", then within P, the names "X", "Y", and "Global_Variable" are all distinct access paths. If Global_Variable's type is neither pass-by-copy nor pass-by-reference, then it is a bounded error to assign to Global_Variable and then read X or Y, since the language does not specify whether the old or the new value would be read. On the other hand, if Global_Variable's type is pass-by-copy, then the old value would always be read, and there is no error. Similarly, if Global_Variable's type is defined by the language to be pass-by-reference, then the new value would always be read, and again there is no error.

12.b **Reason:** We are saying *assign* here, not *update*, because updating any subcomponent is considered to update the enclosing object.

12.c The "still exists" part is so that a read after the subprogram returns is OK.

12.d If the parameter is of a by-copy type, then there is no issue here — the formal is not a view of the actual. If the parameter is of a by-reference type, then the programmer may depend on updates through one access path being visible through some other access path, just as if the parameter were of an access type.

12.e **Implementation Note:** The implementation can keep a copy in a register of a parameter whose parameter-passing mechanism is not specified. If a different access path is used to update the object (creating a bounded error situation), then the implementation can still use the value of the register, even though the in-memory version of the object has been changed. However, to keep the error properly bounded, if the implementation chooses to read the in-memory version, it has to be consistent -- it cannot then assume that something it has proven about the register is true of the memory location. For example, suppose the formal parameter is L, the value of L(6) is now in a register, and L(6) is used in an indexed_component as in "A(L(6)) := 99;", where A has bounds 1..3. If the implementation can prove that the value for L(6) in the register is in the range 1..3, then it need not perform the constraint check if it uses the register value. However, if the memory value of L(6) has been changed to 4, and the implementation uses that memory value, then it had better not alter memory outside of A.

12.f Note that the rule allows the implementation to pass a parameter by reference and then keep just part of it in a register, or, equivalently, to pass part of the parameter by reference and another part by copy.

12.g **Reason:** We do not want to go so far as to say that the mere presence of aliasing is wrong. We wish to be able to write the following sorts of things in standard Ada:

12.h
```
procedure Move ( Source  : in  String;
                 Target  : out String;
                 Drop    : in  Truncation := Error;
                 Justify : in  Alignment  := Left;
                 Pad     : in  Character  := Space);
-- Copies elements from Source to Target (safely if they overlap)
```

12.i This is from the standard string handling package. It would be embarrassing if this couldn't be written in Ada!

12.j The "then" before "read" in the rule implies that the implementation can move a read to an earlier place in the code, but not to a later place after a potentially aliased assignment. Thus, if the subprogram reads one of its parameters into a local variable, and then updates another potentially aliased one, the local copy is safe — it is known to have the old value. For example, the above-mentioned Move subprogram can be implemented by copying Source into a local variable before assigning into Target.

12.k For an assignment_statement assigning one array parameter to another, the implementation has to check which direction to copy at run time, in general, in case the actual parameters are overlapping slices. For example:

```
procedure Copy(X : in out String; Y: String) is
begin
    X := Y;
end Copy;
```
12.l

It would be wrong for the compiler to assume that X and Y do not overlap (unless, of course, it can prove otherwise). 12.m

NOTES

5 A formal parameter of mode **in** is a constant view (see 3.3); it cannot be updated within the subprogram_body. 13

Extensions to Ada 83

{*extensions to Ada 83*} The value of an **out** parameter may be read. An **out** parameter is treated like a declared variable without an explicit initial expression. 13.a

Wording Changes from Ada 83

Discussion of copy-in for parts of out parameters is now covered in 6.4.1, “Parameter Associations”. 13.b

The concept of a by-reference type is new to Ada 95. 13.c

We now cover in a general way in 3.7.2 the rule regarding erroneous execution when a discriminant is changed and one of the parameters depends on the discriminant. 13.d

6.3 Subprogram Bodies

[A subprogram_body specifies the execution of a subprogram.] 1

Syntax

{*AI95-00218-03*} subprogram_body ::= 2/2
```
   [overriding_indicator]
   subprogram_specification is
     declarative_part
   begin
      handled_sequence_of_statements
   end [designator];
```

If a designator appears at the end of a subprogram_body, it shall repeat the defining_designator of the subprogram_specification. 3

Legality Rules

[In contrast to other bodies,] a subprogram_body need not be the completion of a previous declaration[, in which case the body declares the subprogram]. If the body is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a subprogram_body that completes a declaration shall conform fully to that of the declaration. {*full conformance (required)*} 4

Static Semantics

A subprogram_body is considered a declaration. It can either complete a previous declaration, or itself be the initial declaration of the subprogram. 5

Dynamic Semantics

{*elaboration (non-generic subprogram_body)* [partial]} The elaboration of a non-generic subprogram_body has no other effect than to establish that the subprogram can from then on be called without failing the Elaboration_Check. 6

Ramification: See 12.2 for elaboration of a generic body. Note that protected subprogram_bodies never get elaborated; the elaboration of the containing protected_body allows them to be called without failing the Elaboration_Check. 6.a

7 {*execution (subprogram_body)* [partial]} [The execution of a subprogram_body is invoked by a subprogram call.] For this execution the declarative_part is elaborated, and the handled_sequence_of_statements is then executed.

Examples

8 *Example of procedure body:*

9
```
procedure Push(E : in Element_Type; S : in out Stack) is
begin
   if S.Index = S.Size then
      raise Stack_Overflow;
   else
      S.Index := S.Index + 1;
      S.Space(S.Index) := E;
   end if;
end Push;
```

10 *Example of a function body:*

11
```
function Dot_Product(Left, Right : Vector) return Real is
   Sum : Real := 0.0;
begin
   Check(Left'First = Right'First and Left'Last = Right'Last);
   for J in Left'Range loop
      Sum := Sum + Left(J)*Right(J);
   end loop;
   return Sum;
end Dot_Product;
```

Extensions to Ada 83

11.a {*extensions to Ada 83*} A renaming_declaration may be used instead of a subprogram_body.

Wording Changes from Ada 83

11.b The syntax rule for subprogram_body now uses the syntactic category handled_sequence_of_statements.

11.c The declarative_part of a subprogram_body is now required; that doesn't make any real difference, because a declarative_part can be empty.

11.d We have incorporated some rules from RM83-6.5 here.

11.e RM83 forgot to restrict the definition of elaboration of a subprogram_body to non-generics.

Wording Changes from Ada 95

11.f/2 {*AI95-00218-03*} Overriding_indicator is added to subprogram_body.

6.3.1 Conformance Rules

1 {*conformance*} [When subprogram profiles are given in more than one place, they are required to conform in one of four ways: type conformance, mode conformance, subtype conformance, or full conformance.]

Static Semantics

2/1 {*8652/0011*} {*AI95-00117-01*} {*convention*} {*calling convention*} [As explained in B.1, “Interfacing Pragmas”, a *convention* can be specified for an entity.] Unless this International Standard states otherwise, the default convention of an entity is Ada. [For a callable entity or access-to-subprogram type, the convention is called the *calling convention*.] The following conventions are defined by the language:

3
- {*Ada calling convention*} {*calling convention (Ada)*} The default calling convention for any subprogram not listed below is *Ada*. [A pragma Convention, Import, or Export may be used to override the default calling convention (see B.1)].

3.a **Ramification:** See also the rule about renamings-as-body in 8.5.4.

- {*Intrinsic calling convention*} {*calling convention (Intrinsic)*} The *Intrinsic* calling convention represents subprograms that are “built in” to the compiler. The default calling convention is Intrinsic for the following: 4
  - an enumeration literal; 5
  - a "/=" operator declared implicitly due to the declaration of "=" (see 6.6); 6
  - any other implicitly declared subprogram unless it is a dispatching operation of a tagged type; 7
  - an inherited subprogram of a generic formal tagged type with unknown discriminants; 8

Reason: Consider: 8.a.1/1

8.a.2/1
```
package P is
    type Root is tagged null record;
    procedure Proc(X: Root);
end P;
```

8.a.3/1
```
generic
    type Formal(<>) is new Root with private;
package G is
    ...
end G;
```

8.a.4/1
```
package body G is
    ...
    X: Formal := ...;
    ...
    Proc(X); -- This is a dispatching call in Instance, because
             -- the actual type for Formal is class-wide.
    ...
    -- Proc'Access would be illegal here, because it is of
    -- convention Intrinsic, by the above rule.
end G;
```

8.a.5/1
```
type Actual is new Root with ...;
procedure Proc(X: Actual);
package Instance is new G(Formal => Actual'Class);
    -- It is legal to pass in a class-wide actual, because Formal
    -- has unknown discriminants.
```

Within Instance, all calls to Proc will be dispatching calls, so Proc doesn't really exist in machine code, so we wish to avoid taking 'Access of it. This rule applies to those cases where the actual type might be class-wide, and makes these Intrinsic, thus forbidding 'Access. 8.a.6/1

  - an attribute that is a subprogram; 9
  - {*AI95-00252-01*} a subprogram declared immediately within a protected_body; 10/2
  - {*AI95-00252-01*} {*AI95-00407-01*} any prefixed view of a subprogram (see 4.1.3). 10.1/2

Reason: The profile of a prefixed view is different than the “real” profile of the subprogram (it doesn't have the first parameter), so we don't want to be able to take 'Access of it, as that would require generating a wrapper of some sort. 10.a/2

[The Access attribute is not allowed for Intrinsic subprograms.] 11

Ramification: The Intrinsic calling convention really represents any number of calling conventions at the machine code level; the compiler might have a different instruction sequence for each intrinsic. That's why the Access attribute is disallowed. We do not wish to require the implementation to generate an out of line body for an intrinsic. 11.a

Whenever we wish to disallow the Access attribute in order to ease implementation, we make the subprogram Intrinsic. Several language-defined subprograms have “**pragma** Convention(Intrinsic, ...);”. An implementation might actually implement this as “**pragma** Import(Intrinsic, ...);”, if there is really no body, and the implementation of the subprogram is built into the code generator. 11.b

Subprograms declared in protected_bodies will generally have a special calling convention so as to pass along the identification of the current instance of the protected type. The convention is not *protected* since such local subprograms need not contain any “locking” logic since they are not callable via “external” calls; this rule prevents an access value designating such a subprogram from being passed outside the protected unit. 11.c

The “implicitly declared subprogram” above refers to predefined operators (other than the "=" of a tagged type) and the inherited subprograms of untagged types. 11.d

12 • {*protected calling convention*} {*calling convention (protected)*} The default calling convention is *protected* for a protected subprogram, and for an access-to-subprogram type with the reserved word **protected** in its definition.

13 • {*entry calling convention*} {*calling convention (entry)*} The default calling convention is *entry* for an entry.

13.1/2 • {*AI95-00254-01*} {*AI95-00409-01*} The calling convention for an anonymous access-to-subprogram parameter or anonymous access-to-subprogram result is *protected* if the reserved word **protected** appears in its definition and otherwise is the convention of the subprogram that contains the parameter.

13.a/2 **Ramification:** The calling convention for other anonymous access-to-subprogram types is Ada.

13.2/1 • {*8652/0011*} {*AI95-00117-01*} [If not specified above as Intrinsic, the calling convention for any inherited or overriding dispatching operation of a tagged type is that of the corresponding subprogram of the parent type.] The default calling convention for a new dispatching operation of a tagged type is the convention of the type.

13.a.1/1 **Reason:** The first rule is officially stated in 3.9.2. The second is intended to make interfacing to foreign OOP languages easier, by making the default be that the type and operations all have the same convention.

14 Of these four conventions, only Ada and Intrinsic are allowed as a *convention_*identifier in a pragma Convention, Import, or Export.

14.a **Discussion:** The names of the *protected* and *entry* calling conventions cannot be used in the interfacing pragmas. Note that **protected** and **entry** are reserved words.

15/2 {*AI95-00409-01*} {*type conformance*} {*profile (type conformant)*} Two profiles are *type conformant* if they have the same number of parameters, and both have a result if either does, and corresponding parameter and result types are the same, or, for access parameters or access results, corresponding designated types are the same, or corresponding designated profiles are type conformant. {*type profile: See profile, type conformant*}

15.a/2 **Discussion:** {*AI95-00409-01*} For anonymous access-to-object parameters, the designated types have to be the same for type conformance, not the access types, since in general each access parameter has its own anonymous access type, created when the subprogram is called. Of course, corresponding parameters have to be either both access parameters or both not access parameters.

15.b/2 {*AI95-00409-01*} Similarly, for anonymous access-to-subprogram parameters, the designated profiles of the types, not the types themselves, have to be conformant.

16/2 {*AI95-00318-02*} {*AI95-00409-01*} {*mode conformance*} {*profile (mode conformant)*} Two profiles are *mode conformant* if they are type-conformant, and corresponding parameters have identical modes, and, for access parameters or access result types, the designated subtypes statically match, or the designated profiles are subtype conformant. {*statically matching (required)* [partial]}

17 {*subtype conformance*} {*profile (subtype conformant)*} Two profiles are *subtype conformant* if they are mode-conformant, corresponding subtypes of the profile statically match, and the associated calling conventions are the same. The profile of a generic formal subprogram is not subtype-conformant with any other profile. {*statically matching (required)* [partial]}

17.a **Ramification:** {*generic contract issue* [partial]}

18 {*full conformance (for profiles)*} {*profile (fully conformant)*} Two profiles are *fully conformant* if they are subtype-conformant, and corresponding parameters have the same names and have default_expressions that are fully conformant with one another.

18.a **Ramification:** Full conformance requires subtype conformance, which requires the same calling conventions. However, the calling convention of the declaration and body of a subprogram or entry are always the same by definition.

{*full conformance (for expressions)*} Two expressions are *fully conformant* if, [after replacing each use of an operator with the equivalent function_call:] 19

- each constituent construct of one corresponds to an instance of the same syntactic category in the other, except that an expanded name may correspond to a direct_name (or character_literal) or to a different expanded name in the other; and 20
- each direct_name, character_literal, and selector_name that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding direct_name, character_literal, or selector_name in the other; and 21

  Ramification: Note that it doesn't say "respectively" because a direct_name can correspond to a selector_name, and vice-versa, by the previous bullet. This rule allows the prefix of an expanded name to be removed, or replaced with a different prefix that denotes a renaming of the same entity. However, it does not allow a direct_name or selector_name to be replaced with one denoting a distinct renaming (except for direct_names and selector_names in prefixes of expanded names). Note that calls using operator notation are equivalent to calls using prefix notation. 21.a

  Given the following declarations: 21.b

  21.c
  ```
  package A is
      function F(X : Integer := 1) return Boolean;
  end A;
  ```

  21.d
  ```
  with A;
  package B is
      package A_View renames A;
      function F_View(X : Integer := 9999) return Boolean renames F;
  end B;
  ```

  21.e
  ```
  with A, B; use A, B;
  procedure Main is ...
  ```

  Within Main, the expressions "F", "A.F", "B.A_View.F", and "A_View.F" are all fully conformant with one another. However, "F" and "F_View" are not fully conformant. If they were, it would be bad news, since the two denoted views have different default_expressions. 21.f

- {*8652/0018*} {*AI95-00175-01*} each attribute_designator in one must be the same as the corresponding attribute_designator in the other; and 21.1/1
- each primary that is a literal in one has the same value as the corresponding literal in the other. 22

  Ramification: The literals may be written differently. 22.a

  Ramification: Note that the above definition makes full conformance a transitive relation. 22.b

{*full conformance (for known_discriminant_parts)*} Two known_discriminant_parts are *fully conformant* if they have the same number of discriminants, and discriminants in the same positions have the same names, statically matching subtypes, and default_expressions that are fully conformant with one another. {*statically matching (required)* [partial]} 23

{*full conformance (for discrete_subtype_definitions)*} Two discrete_subtype_definitions are *fully conformant* if they are both subtype_indications or are both ranges, the subtype_marks (if any) denote the same subtype, and the corresponding simple_expressions of the ranges (if any) fully conform. 24

Ramification: In the subtype_indication case, any ranges have to *be* corresponding; that is, two subtype_indications cannot conform unless both or neither has a range. 24.a

Discussion: This definition is used in 9.5.2, "Entries and Accept Statements" for the conformance required between the discrete_subtype_definitions of an entry_declaration for a family of entries and the corresponding entry_index_specification of the entry_body. 24.b

{*AI95-00345-01*} {*AI95-00397-01*} {*prefixed view profile*} The *prefixed view profile* of a subprogram is the profile obtained by omitting the first parameter of that subprogram. There is no prefixed view profile for a parameterless subprogram. For the purposes of defining subtype and mode conformance, the convention of a prefixed view profile is considered to match that of either an entry or a protected operation. 24.1/2

Discussion: This definition is used to define how primitive subprograms of interfaces match operations in task and protected type definitions (see 9.1 and 9.4). 24.c/2

24.d/2 **Reason:** The weird rule about conventions is pretty much required for synchronized interfaces to make any sense. There will be wrappers all over the place for interfaces anyway. Of course, this doesn't imply that entries have the same convention as protected operations.

Implementation Permissions

25 An implementation may declare an operator declared in a language-defined library unit to be intrinsic.

Extensions to Ada 83

25.a {*extensions to Ada 83*} The rules for full conformance are relaxed — they are now based on the structure of constructs, rather than the sequence of lexical elements. This implies, for example, that "(X, Y: T)" conforms fully with "(X: T; Y: T)", and "(X: T)" conforms fully with "(X: **in** T)".

Wording Changes from Ada 95

25.b/2 {*8652/0011*} {*AI95-00117-01*} **Corrigendum:** Clarified that the default convention is Ada. Also clarified that the convention of a primitive operation of a tagged type is the same as that of the type.

25.c/2 {*8652/0018*} {*AI95-00175-01*} **Corrigendum:** Added wording to ensure that two attributes conform only if they have the same attribute_designator.

25.d/2 {*AI95-00252-01*} {*AI95-00254-01*} {*AI95-00407-01*} Defined the calling convention for anonymous access-to-subprogram types and for prefixed views of subprograms (see 4.1.3).

25.e/2 {*AI95-00318-02*} Defined the conformance of access result types (see 6.1).

25.f/2 {*AI95-00345-01*} {*AI95-00397-01*} Defined the prefixed view profile of subprograms for later use.

25.g/2 {*AI95-00409-01*} Defined the conformance of anonymous access-to-subprogram parameters.

6.3.2 Inline Expansion of Subprograms

1 [Subprograms may be expanded in line at the call site.]

Syntax

2 {*program unit pragma (Inline)* [partial]} {*pragma, program unit (Inline)* [partial]} The form of a pragma Inline, which is a program unit pragma (see 10.1.5), is as follows:

3 **pragma** Inline(name {, name});

Legality Rules

4 The pragma shall apply to one or more callable entities or generic subprograms.

Static Semantics

5 If a pragma Inline applies to a callable entity, this indicates that inline expansion is desired for all calls to that entity. If a pragma Inline applies to a generic subprogram, this indicates that inline expansion is desired for all calls to all instances of that generic subprogram.

5.a **Ramification:** Note that inline expansion is desired no matter what name is used in the call. This allows one to request inlining for only one of several overloaded subprograms as follows:

5.b
```
package IO is
   procedure Put(X : in Integer);
   procedure Put(X : in String);
   procedure Put(X : in Character);
private
   procedure Character_Put(X : in Character) renames Put;
   pragma Inline(Character_Put);
end IO;
```

5.c
```
with IO; use IO;
procedure Main is
   I : Integer;
   C : Character;
begin
   ...
   Put(C); -- Inline expansion is desired.
   Put(I); -- Inline expansion is NOT desired.
end Main;
```

Ramification: The meaning of a subprogram can be changed by a pragma Inline only in the presence of failing checks (see 11.6). 5.d

Implementation Permissions

For each call, an implementation is free to follow or to ignore the recommendation expressed by the pragma. 6

Ramification: Note, in particular, that the recommendation cannot always be followed for a recursive call, and is often infeasible for entries. Note also that the implementation can inline calls even when no such desire was expressed by a pragma, so long as the semantics of the program remains unchanged. 6.a

{*AI95-00309-01*} An implementation may allow a pragma Inline that has an argument which is a direct_name denoting a subprogram_body of the same declarative_part. 6.1/2

Reason: This is allowed for Ada 83 compatibility. This is only a permission as this usage is considered obsolescent. 6.b/2

Discussion: We only need to allow this in declarative_parts, because a body is only allowed in another body, and these all have declarative_parts. 6.c/2

NOTES

6 The name in a pragma Inline can denote more than one entity in the case of overloading. Such a pragma applies to all of the denoted entities. 7

Incompatibilities With Ada 83

{*AI95-00309-01*} {*incompatibilities with Ada 83*} A pragma Inline cannot refer to a subprogram_body outside of that body. The pragma can be given inside of the subprogram body. Ada 2005 adds an Implementation Permission to allow this usage for compatibility (and Ada 95 implementations also can use this permission), but implementations do not have to allow such pragmas. 7.a/2

Extensions to Ada 83

{*extensions to Ada 83*} A pragma Inline is allowed inside a subprogram_body if there is no corresponding subprogram_declaration. This is for uniformity with other program unit pragmas. 7.b

Extensions to Ada 95

{*AI95-00309-01*} {*extensions to Ada 95*} **Amendment Correction:** Implementations are allowed to let Pragma Inline apply to a subprogram_body. 7.c/2

6.4 Subprogram Calls

{*subprogram call*} A *subprogram call* is either a procedure_call_statement or a function_call; [it invokes the execution of the subprogram_body. The call specifies the association of the actual parameters, if any, with formal parameters of the subprogram.] 1

Syntax

procedure_call_statement ::= 2
  *procedure*_name;
 | *procedure*_prefix actual_parameter_part;

function_call ::= 3
  *function*_name
 | *function*_prefix actual_parameter_part

4 actual_parameter_part ::=
(parameter_association {, parameter_association})

5 parameter_association ::=
[*formal_parameter*_selector_name =>] explicit_actual_parameter

6 explicit_actual_parameter ::= expression | *variable*_name

7 {*named association*} {*positional association*} A parameter_association is *named* or *positional* according to whether or not the *formal_parameter*_selector_name is specified. Any positional associations shall precede any named associations. Named associations are not allowed if the prefix in a subprogram call is an attribute_reference.

7.a **Ramification:** This means that the formal parameter names used in describing predefined attributes are to aid presentation of their semantics, but are not intended for use in actual calls.

Name Resolution Rules

8/2 {*AI95-00310-01*} The name or prefix given in a procedure_call_statement shall resolve to denote a callable entity that is a procedure, or an entry renamed as (viewed as) a procedure. The name or prefix given in a function_call shall resolve to denote a callable entity that is a function. The name or prefix shall not resolve to denote an abstract subprogram unless it is also a dispatching subprogram. [When there is an actual_parameter_part, the prefix can be an implicit_dereference of an access-to-subprogram value.]

8.a.1/2 **Discussion:** {*AI95-00310-01*} This rule is talking about dispatching operations (which is a static concept) and not about dispatching calls (which is a dynamic concept).

8.a **Ramification:** The function can be an operator, enumeration literal, attribute that is a function, etc.

9 A subprogram call shall contain at most one association for each formal parameter. Each formal parameter without an association shall have a default_expression (in the profile of the view denoted by the name or prefix). This rule is an overloading rule (see 8.6).

Dynamic Semantics

10/2 {*AI95-00345-01*} {*execution (subprogram call)* [partial]} For the execution of a subprogram call, the name or prefix of the call is evaluated, and each parameter_association is evaluated (see 6.4.1). If a default_-expression is used, an implicit parameter_association is assumed for this rule. These evaluations are done in an arbitrary order. The subprogram_body is then executed, or a call on an entry or protected subprogram is performed (see 3.9.2). Finally, if the subprogram completes normally, then after it is left, any necessary assigning back of formal to actual parameters occurs (see 6.4.1).

10.a **Discussion:** The implicit association for a default is only for this run-time rule. At compile time, the visibility rules are applied to the default at the place where it occurs, not at the place of a call.

10.b **To be honest:** If the subprogram is inherited, see 3.4, “Derived Types and Classes”.

10.c If the subprogram is protected, see 9.5.1, “Protected Subprograms and Protected Actions”.

10.d If the subprogram is really a renaming of an entry, see 9.5.3, “Entry Calls”.

10.d.1/2 {*AI95-00345-01*} If the subprogram is implemented by an entry or protected subprogram, it will be treated as a dispatching call to the corresponding entry (see 9.5.3, “Entry Calls”) or protected subprogram (see 9.5.1, “Protected Subprograms and Protected Actions”).

10.e/2 {*AI95-00348-01*} Normally, the subprogram_body that is executed by the above rule is the one for the subprogram being called. For an enumeration literal, implicitly declared (but noninherited) subprogram, null procedure, or an attribute that is a subprogram, an implicit body is assumed. For a dispatching call, 3.9.2, “Dispatching Operations of Tagged Types” defines which subprogram_body is executed.

10.1/2 {*AI95-00407-01*} If the name or prefix of a subprogram call denotes a prefixed view (see 4.1.3), the subprogram call is equivalent to a call on the underlying subprogram, with the first actual parameter being provided by the prefix of the prefixed view (or the Access attribute of this prefix if the first formal parameter is an access parameter), and the remaining actual parameters given by the actual_parameter_part, if any.

{*AI95-00318-02*} {*Program_Error (raised by failure of run-time check)*} The exception Program_Error is raised at the point of a function_call if the function completes normally without executing a return statement. 11/2

> **Discussion:** We are committing to raising the exception at the point of call, for uniformity — see AI83-00152. This happens after the function is left, of course. 11.a

> Note that there is no name for suppressing this check, since the check imposes no time overhead and minimal space overhead (since it can usually be statically eliminated as dead code). 11.b

{*AI95-00231-01*} A function_call denotes a constant, as defined in 6.5; the nominal subtype of the constant is given by the nominal subtype of the function result. {*nominal subtype (of the result of a function_call)* [partial]} {*constant (result of a function_call)* [partial]} 12/2

Examples

Examples of procedure calls: 13

14
```
Traverse_Tree;                                               -- see 6.1
Print_Header(128, Title, True);                              -- see 6.1
```

15
```
Switch(From => X, To => Next);                               -- see 6.1
Print_Header(128, Header => Title, Center => True);          -- see 6.1
Print_Header(Header => Title, Center => True, Pages => 128); -- see 6.1
```

Examples of function calls: 16

17
```
Dot_Product(U, V)   -- see 6.1 and 6.3
Clock               -- see 9.6
F.all               -- presuming F is of an access-to-subprogram type — see 3.10
```

Examples of procedures with default expressions: 18

19
```
procedure Activate(Process : in Process_Name;
                   After   : in Process_Name := No_Process;
                   Wait    : in Duration := 0.0;
                   Prior   : in Boolean := False);
```

20
```
procedure Pair(Left, Right : in Person_Name := new Person);   -- see 3.10.1
```

Examples of their calls: 21

22
```
Activate(X);
Activate(X, After => Y);
Activate(X, Wait => 60.0, Prior => True);
Activate(X, Y, 10.0, False);
```

23
```
Pair;
Pair(Left => new Person, Right => new Person);
```

NOTES

7 If a default_expression is used for two or more parameters in a multiple parameter_specification, the default_expression is evaluated once for each omitted parameter. Hence in the above examples, the two calls of Pair are equivalent. 24

Examples

Examples of overloaded subprograms: 25

26
```
procedure Put(X : in Integer);
procedure Put(X : in String);
```

27
```
procedure Set(Tint   : in Color);
procedure Set(Signal : in Light);
```

28 *Examples of their calls:*

29
```
Put(28);
Put("no possible ambiguity here");
```

30
```
Set(Tint   => Red);
Set(Signal => Red);
Set(Color'(Red));
```

31
```
-- Set(Red) would be ambiguous since Red may
-- denote a value either of type Color or of type Light
```

Wording Changes from Ada 83

31.a We have gotten rid of parameters "of the form of a type conversion" (see RM83-6.4.1(3)). The new view semantics of type_conversions allows us to use normal type_conversions instead.

31.b We have moved wording about run-time semantics of parameter associations to 6.4.1.

31.c We have moved wording about raising Program_Error for a function that falls off the end to here from RM83-6.5.

Extensions to Ada 95

31.d/2 {*AI95-00310-01*} {*extensions to Ada 95*} Nondispatching abstract operations are no longer considered when resolving a subprogram call. That makes it possible to use **abstract** to "undefine" a predefined operation for an untagged type. That's especially helpful when defining custom arithmetic packages.

Wording Changes from Ada 95

31.e/2 {*AI95-00231-01*} Changed the definition of the nominal subtype of a function_call to use the nominal subtype wording of 6.1, to take into account null_exclusions and access result types.

31.f/2 {*AI95-00345-01*} Added wording to clarify that the meaning of a call on a subprogram "implemented by" an entry or protected operation is defined by 3.9.2.

31.g/2 {*AI95-00407-01*} Defined the meaning of a call on a prefixed view of a subprogram (see 4.1.3).

6.4.1 Parameter Associations

1 [{*parameter passing*} A parameter association defines the association between an actual parameter and a formal parameter.]

Language Design Principles

1.a The parameter passing rules for **out** parameters are designed to ensure that the parts of a type that have implicit initial values (see 3.3.1) don't become "de-initialized" by being passed as an **out** parameter.

Name Resolution Rules

2 The *formal_parameter_*selector_name of a parameter_association shall resolve to denote a parameter_specification of the view being called.

3 {*actual parameter (for a formal parameter)*} The *actual parameter* is either the explicit_actual_parameter given in a parameter_association for a given formal parameter, or the corresponding default_expression if no parameter_association is given for the formal parameter. {*expected type (actual parameter)*} The expected type for an actual parameter is the type of the corresponding formal parameter.

3.a **To be honest:** The corresponding default_expression is the one of the corresponding formal parameter in the profile of the view denoted by the name or prefix of the call.

4 If the mode is **in**, the actual is interpreted as an expression; otherwise, the actual is interpreted only as a name, if possible.

4.a **Ramification:** This formally resolves the ambiguity present in the syntax rule for explicit_actual_parameter. Note that we don't actually require that the actual be a name if the mode is not **in**; we do that below.

Legality Rules

If the mode is **in out** or **out**, the actual shall be a name that denotes a variable. 5

Discussion: We no longer need "or a type_conversion whose argument is the name of a variable," because a type_conversion is now a name, and a type_conversion of a variable is a variable. 5.a

Reason: The requirement that the actual be a (variable) name is not an overload resolution rule, since we don't want the difference between expression and name to be used to resolve overloading. For example: 5.b

5.c
```
procedure Print(X : in Integer; Y : in Boolean := True);
procedure Print(Z : in out Integer);
. . .
Print(3); -- Ambiguous!
```

The above call to Print is ambiguous even though the call is not compatible with the second Print which requires an actual that is a (variable) name ("3" is an expression, not a name). This requirement is a legality rule, so overload resolution fails before it is considered, meaning that the call is ambiguous. 5.d

The type of the actual parameter associated with an access parameter shall be convertible (see 4.6) to its anonymous access type. {*convertible (required)* [partial]} 6

Dynamic Semantics

{*evaluation (parameter_association)* [partial]} For the evaluation of a parameter_association: 7

- The actual parameter is first evaluated. 8
- For an access parameter, the access_definition is elaborated, which creates the anonymous access type. 9
- For a parameter [(of any mode)] that is passed by reference (see 6.2), a view conversion of the actual parameter to the nominal subtype of the formal parameter is evaluated, and the formal parameter denotes that conversion. {*implicit subtype conversion (parameter passing)* [partial]} 10

Discussion: We are always allowing sliding, even for [**in**[**out** by-reference parameters. 10.a

- {*assignment operation (during evaluation of a parameter_association)*} For an **in** or **in out** parameter that is passed by copy (see 6.2), the formal parameter object is created, and the value of the actual parameter is converted to the nominal subtype of the formal parameter and assigned to the formal. {*implicit subtype conversion (parameter passing)* [partial]} 11

Ramification: The conversion mentioned here is a value conversion. 11.a

- For an **out** parameter that is passed by copy, the formal parameter object is created, and: 12
  - For an access type, the formal parameter is initialized from the value of the actual, without a constraint check; 13

Reason: This preserves the Language Design Principle that an object of an access type is always initialized with a "reasonable" value. 13.a

  - For a composite type with discriminants or that has implicit initial values for any subcomponents (see 3.3.1), the behavior is as for an **in out** parameter passed by copy. 14

Reason: This ensures that no part of an object of such a type can become "de-initialized" by being part of an **out** parameter. 14.a

Ramification: This includes an array type whose component type is an access type, and a record type with a component that has a default_expression, among other things. 14.b

  - For any other type, the formal parameter is uninitialized. If composite, a view conversion of the actual parameter to the nominal subtype of the formal is evaluated [(which might raise Constraint_Error)], and the actual subtype of the formal is that of the view conversion. If elementary, the actual subtype of the formal is given by its nominal subtype. 15

15.a **Ramification:** This case covers scalar types, and composite types whose subcomponent's subtypes do not have any implicit initial values. The view conversion for composite types ensures that if the lengths don't match between an actual and a formal array parameter, the Constraint_Error is raised before the call, rather than after.

16 {*constrained (object)* [partial]} {*unconstrained (object)* [partial]} A formal parameter of mode **in out** or **out** with discriminants is constrained if either its nominal subtype or the actual parameter is constrained.

17 {*parameter copy back*} {*copy back of parameters*} {*parameter assigning back*} {*assigning back of parameters*} {*assignment operation (during parameter copy back)*} After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by copy, the value of the formal parameter is converted to the subtype of the variable given as the actual parameter and assigned to it. {*implicit subtype conversion (parameter passing)* [partial]} These conversions and assignments occur in an arbitrary order.

17.a **Ramification:** The conversions mentioned above during parameter passing might raise Constraint_Error — (see 4.6).

17.b **Ramification:** If any conversion or assignment as part of parameter passing propagates an exception, the exception is raised at the place of the subprogram call; that is, it cannot be handled inside the subprogram_body.

17.c **Proof:** Since these checks happen before or after executing the subprogram_body, the execution of the subprogram_body does not dynamically enclose them, so it can't handle the exceptions.

17.d **Discussion:** The variable we're talking about is the one denoted by the *variable_*name given as the explicit_actual_parameter. If this *variable_*name is a type_conversion, then the rules in 4.6 for assigning to a view conversion apply. That is, if X is of subtype S1, and the actual is S2(X), the above-mentioned conversion will convert to S2, and the one mentioned in 4.6 will convert to S1.

Extensions to Ada 83

17.e {*extensions to Ada 83*} In Ada 95, a program can rely on the fact that passing an object as an **out** parameter does not "de-initialize" any parts of the object whose subtypes have implicit initial values. (This generalizes the RM83 rule that required copy-in for parts that were discriminants or of an access type.)

Wording Changes from Ada 83

17.f We have eliminated the subclause on Default Parameters, as it is subsumed by earlier clauses and subclauses.

6.5 Return Statements

1/2 {*AI95-00318-02*} A simple_return_statement or extended_return_statement (collectively called a *return statement*) {*return statement*} is used to complete the execution of the innermost enclosing subprogram_body, entry_body, or accept_statement.

Syntax

2/2 {*AI95-00318-02*} simple_return_statement ::= **return** [expression];

2.1/2 {*AI95-00318-02*} extended_return_statement ::=
  return defining_identifier : [**aliased**] return_subtype_indication [:= expression] [**do**
    handled_sequence_of_statements
  end return];

2.2/2 {*AI95-00318-02*} return_subtype_indication ::= subtype_indication | access_definition

Name Resolution Rules

3/2 {*AI95-00318-02*} {*result subtype (of a function)*} The *result subtype* of a function is the subtype denoted by the subtype_mark, or defined by the access_definition, after the reserved word **return** in the profile of the function.{*expected type (expression of simple_* [partial]} The expected type for the expression, if any, of a simple_return_statement is the result type of the corresponding function. {*expected type (expression of extended_return_statement)* [partial]} The expected type for the expression of an extended_return_statement is that of the return_subtype_indication.

3.a **To be honest:** The same applies to generic functions.

Legality Rules

{*AI95-00318-02*} {*apply (to a callable construct by a return statement)*} A return statement shall be within a callable construct, and it *applies to* the innermost callable construct or extended_return_statement that contains it. A return statement shall not be within a body that is within the construct to which the return statement applies. 4/2

{*AI95-00318-02*} A function body shall contain at least one return statement that applies to the function body, unless the function contains code_statements. A simple_return_statement shall include an expression if and only if it applies to a function body. An extended_return_statement shall apply to a function body. 5/2

> **Reason:** {*AI95-00318-02*} The requirement that a function body has to have at least one return statement is a "helpful" restriction. There has been some interest in lifting this restriction, or allowing a raise statement to substitute for the return statement. However, there was enough interest in leaving it as is that we decided not to change it. 5.a/2

> **Ramification:** {*AI95-00318-02*} A return statement can apply to an extended_return_statement, so a simple_-return_statement without an expression can be given in one. However, neither simple_return_statement with an expression nor an extended_return_statement can be given inside an extended_return_statement, as they must apply (directly) to a function body. 5.b/2

{*AI95-00318-02*} For an extended_return_statement that applies to a function body: 5.1/2

- {*AI95-00318-02*} If the result subtype of the function is defined by a subtype_mark, the return_-subtype_indication shall be a subtype_indication. The type of the subtype_indication shall be the result type of the function. If the result subtype of the function is constrained, then the subtype defined by the subtype_indication shall also be constrained and shall statically match this result subtype. {*statically matching (required)* [partial]} If the result subtype of the function is unconstrained, then the subtype defined by the subtype_indication shall be a definite subtype, or there shall be an expression. 5.2/2
- {*AI95-00318-02*} If the result subtype of the function is defined by an access_definition, the return_subtype_indication shall be an access_definition. The subtype defined by the access_definition shall statically match the result subtype of the function. The accessibility level of this anonymous access subtype is that of the result subtype. 5.3/2

{*AI95-00318-02*} For any return statement that applies to a function body: 5.4/2

- {*AI95-00318-02*} If the result subtype of the function is limited, then the expression of the return statement (if any) shall be an aggregate, a function call (or equivalent use of an operator), or a qualified_expression or parenthesized expression whose operand is one of these. 5.5/2

  > **Discussion:** In other words, if limited, the expression must produce a "new" object, rather than being the name of a preexisting object (which would imply copying). 5.c/2

- {*AI95-00416-01*} If the result subtype of the function is class-wide, the accessibility level of the type of the expression of the return statement shall not be statically deeper than that of the master that elaborated the function body. If the result subtype has one or more unconstrained access discriminants, the accessibility level of the anonymous access type of each access discriminant, as determined by the expression of the simple_return_statement or the return_subtype_-indication, shall not be statically deeper than that of the master that elaborated the function body. 5.6/2

  > **Discussion:** We know that if the result type is class wide, then there must be an expression of the return statement. Similarly, if the result subtype is unconstrained, then either the return_subtype_indication (if any) is constrained, or there must be an expression. 5.d/2

Static Semantics

{*AI95-00318-02*} {*return object (extended_return_statement)* [partial]} Within an extended_return_statement, the *return object* is declared with the given defining_identifier, with the nominal subtype defined by the return_subtype_indication. 5.7/2

Dynamic Semantics

5.8/2 {*AI95-00318-02*} {*AI95-00416-01*} {*execution (extended_return_statement)* [partial]} For the execution of an extended_return_statement, the subtype_indication or access_definition is elaborated. This creates the nominal subtype of the return object. If there is an expression, it is evaluated and converted to the nominal subtype (which might raise Constraint_Error — see 4.6{*implicit subtype conversion (function return)* [partial]}); the return object is created and the converted value is assigned to the return object. Otherwise, the return object is created and initialized by default as for a stand-alone object of its nominal subtype (see 3.3.1). If the nominal subtype is indefinite, the return object is constrained by its initial value.{*creation (of a return object)* [partial]}

5.e/2 **Ramification:** If the result type is controlled or has a controlled part, appropriate calls on Initialize or Adjust are performed prior to executing the handled_sequence_of_statements, except when the initial expression is an aggregate (which requires build-in-place with no call on Adjust).

5.f/2 If the return statement is left without resulting in a return (for example, due to an exception propagated from the expression or the handled_sequence_of_statements, or a goto out of the handled_sequence_of_statements), the return object is finalized prior to leaving the return statement.

6/2 {*AI95-00318-02*} {*execution (simple_* [partial]} For the execution of a simple_return_statement, the expression (if any) is first evaluated, converted to the result subtype, and then is assigned to the anonymous *return object*. {*return object (simple_* [partial]} {*implicit subtype conversion (function return)* [partial]}

6.a **Ramification:** The conversion might raise Constraint_Error — (see 4.6).

7/2 {*AI95-00318-02*} {*AI95-00416-01*} [If the return object has any parts that are tasks, the activation of those tasks does not occur until after the function returns (see 9.2).]

7.a/2 **Proof:** This is specified by the rules in 9.2.

7.b/2 **Reason:** Only the caller can know when task activations should take place, as it depends on the context of the call. If the function is being used to initialize the component of some larger object, then that entire object must be initialized before any task activations. Even after the outer object is fully initialized, task activations are still postponed until the **begin** at the end of the declarative part if the function is being used to initialize part of a declared object.

8/2 {*AI95-00318-02*} {*AI95-00344-01*} If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the value of the expression. A check is made that the accessibility level of the type identified by the tag of the result is not deeper than that of the master that elaborated the function body. If this check fails, Program_Error is raised.{*Program_Error (raised by failure of run-time check)*} {*Accessibility_Check* [partial]} {*check, language-defined (Accessibility_Check)*}

8.a/2 **Ramification:** {*AI95-00318-02*} The first sentence is true even if the tag of the expression is different, which could happen if the expression were a view conversion or a dereference of an access value. Note that for a limited type, because of the restriction to aggregates and function calls (and no conversions), the tag will already match.

8.b/2 **Reason:** {*AI95-00318-02*} The first rule ensures that a function whose result type is a specific tagged type always returns an object whose tag is that of the result type. This is important for dispatching on controlling result, and allows the caller to allocate the appropriate amount of space to hold the value being returned (assuming there are no discriminants).

8.c/2 The check prevents the returned object from outliving its type. Note that this check cannot fail for a specific tagged type, as the tag represents the function's type, which necessarily must be declared outside of the function.

Paragraphs 9 through 20 were deleted.

21/2 {*AI95-00318-02*} {*AI95-00402-01*} {*AI95-00416-01*} If the result subtype of a function has one or more unconstrained access discriminants, a check is made that the accessibility level of the anonymous access type of each access discriminant, as determined by the expression or the return_subtype_indication of the function, is not deeper than that of the master that elaborated the function body. If this check fails, Program_Error is raised. {*Program_Error (raised by failure of run-time check)*} {*Accessibility_Check* [partial]} {*check, language-defined (Accessibility_Check)*}

This paragraph was deleted. 21.a/2

Reason: The check prevents the returned object (for a nonlimited type) from outliving the object designated by one of its discriminants. The check is made on the values of the discriminants, which may come from the return_subtype_indication (if constrained), or the expression, but it is never necessary to check both. 21.b/2

{*AI95-00318-02*} For the execution of an extended_return_statement, the handled_sequence_of_statements is executed. Within this handled_sequence_of_statements, the execution of a simple_return_statement that applies to the extended_return_statement causes a transfer of control that completes the extended_return_statement. Upon completion of a return statement that applies to a callable construct, a transfer of control is performed which completes the execution of the callable construct, and returns to the caller. 22/2

{*AI95-00318-02*} In the case of a function, the function_call denotes a constant view of the return object. 23/2

Implementation Permissions

{*AI95-00416-01*} If the result subtype of a function is unconstrained, and a call on the function is used to provide the initial value of an object with a constrained nominal subtype, Constraint_Error may be raised at the point of the call (after abandoning the execution of the function body) if, while elaborating the return_subtype_indication or evaluating the expression of a return statement that applies to the function body, it is determined that the value of the result will violate the constraint of the subtype of this object. 24/2

Reason: Without such a permission, it would be very difficult to implement “build-in-place” semantics. Such an exception is not handleable within the function, because in the return-by-copy case, the constraint check to verify that the result satisfies the constraints of the object being initialized happens after the function returns, and we want the semantics to change as little as possible when switching between return-by-copy and build-in-place. This implies further that upon detecting such a situation, the implementation may need to simulate a goto to a point outside any local exception handlers prior to raising the exception. 24.a/2

Ramification: This permission is allowed during the evaluation of the expression of an extended_return_statement, because the return_subtype_indication may be unconstrained and the expression then would provide the constraints. 24.b/2

Examples

Examples of return statements: 25

26/2
```
{AI95-00318-02} return;                         -- in a procedure body, entry_body,
                                   -- accept_statement, or extended_return_statement
```

27
```
return Key_Value(Last_Index);      -- in a function body
```

28/2
```
{AI95-00318-02} return Node : Cell do           -- in a function body, see 3.10.1 for Cell
   Node.Value := Result;
   Node.Succ := Next_Node;
end return;
```

Incompatibilities With Ada 83

{*AI95-00318-02*} {*incompatibilities with Ada 83*} In Ada 95, if the result type of a function has a part that is a task, then an attempt to return a local variable will raise Program_Error. This is illegal in Ada 2005, see below. In Ada 83, if a function returns a local variable containing a task, execution is erroneous according to AI83-00867. However, there are other situations where functions that return tasks (or that return a variant record only one of whose variants includes a task) are correct in Ada 83 but will raise Program_Error according to the new rules. 28.a/2

The rule change was made because there will be more types (protected types, limited controlled types) in Ada 95 for which it will be meaningless to return a local variable, and making all of these erroneous is unacceptable. The current rule was felt to be the simplest that kept upward incompatibilities to situations involving returning tasks, which are quite rare. 28.b

Wording Changes from Ada 83

This clause has been moved here from chapter 5, since it has mainly to do with subprograms. 28.c

A function now creates an anonymous object. This is necessary so that controlled types will work. 28.d

{*AI95-00318-02*} We have clarified that a return statement applies to a callable construct, not to a callable entity. 28.e/2

28.f/2 {*AI95-00318-02*} There is no need to mention generics in the rules about where a return statement can appear and what it applies to; the phrase "body of a subprogram or generic subprogram" is syntactic, and refers exactly to "subprogram_body".

Incompatibilities With Ada 95

28.g/2 {*AI95-00318-02*} {*incompatibilities with Ada 95*} The entire business about return-by-reference types has been dropped. Instead, the expression of a return statement of a limited type can only be an aggregate or function_call (see 7.5). This means that returning a global object or type_conversion, legal in Ada 95, is now illegal. Such functions can be converted to use anonymous access return types by adding **access** in the function definition and return statement, adding .**all** in uses, and adding **aliased** in the object declarations. This has the advantage of making the reference return semantics much clearer to the casual reader.

28.h/2 We changed these rules so that functions, combined with the new rules for limited types (7.5), can be used as build-in-place constructors for limited types. This reduces the differences between limited and nonlimited types, which will make limited types useful in more circumstances.

Extensions to Ada 95

28.i/2 {*AI95-00318-02*} {*extensions to Ada 95*} The extended_return_statement is new. This provides a name for the object being returned, which reduces the copying needed to return complex objects (including no copying at all for limited objects). It also allows component-by-component construction of the return object.

Wording Changes from Ada 95

28.j/2 {*AI95-00318-02*} The wording was updated to support anonymous access return subtypes.

28.k/2 {*AI95-00318-02*} The term "return expression" was dropped because reviewers found it confusing when applied to the default expression of an extended_return_statement.

28.l/2 {*AI95-00344-01*} {*AI95-00416-01*} Added accessibility checks to class-wide return statements. These checks could not fail in Ada 95 (as all of the types had to be declared at the same level, so the tagged type would necessarily have been at the same level as the type of the object).

28.m/2 {*AI95-00402-01*} {*AI95-00416-01*} Added accessibility checks to return statements for types with access discriminants. Since such types have to be limited in Ada 95, the expression of a return statement would have been illegal in order for this check to fail.

28.n/2 {*AI95-00416-01*} Added an Implementation Permission allowing early raising of Constraint_Error if the result cannot fit in the ultimate object. This gives implementations more flexibility to do built-in-place returns, and is essential for limited types (which cannot be built in a temporary).

6.5.1 Pragma No_Return

1/2 {*AI95-00329-01*} {*AI95-00414-01*} A pragma No_Return indicates that a procedure cannot return normally[; it may propagate an exception or loop forever].

1.a/2 **Discussion:** Pragma No_Deposit will have to wait for Ada 2017. :-)

Syntax

2/2 {*AI95-00329-01*} {*AI95-00414-01*} The form of a pragma No_Return, which is a representation pragma (see 13.1), is as follows:

3/2 **pragma** No_Return(*procedure_*local_name{, *procedure_*local_name});

Legality Rules

4/2 {*AI95-00329-01*} {*AI95-00414-01*} {*non-returning*} Each *procedure_*local_name shall denote one or more procedures or generic procedures; the denoted entities are *non-returning*. The *procedure_*local_name shall not denote a null procedure nor an instance of a generic unit.

4.a/2 **Reason:** A null procedure cannot have the appropriate non-returning semantics, as it does not raise an exception or loop forever.

4.b/2 **Ramification:** The procedure can be abstract. The denoted declaration can be a renaming_declaration if it obeys the usual rules for representation pragmas: the renaming has to occur immediately within the same declarative region as the renamed subprogram. If a non-returning procedure is renamed (anywhere) calls through the new name still have the non-returning semantics.

{*AI95-00329-01*} {*AI95-00414-01*} A return statement shall not apply to a non-returning procedure or generic procedure. 5/2

{*AI95-00414-01*} A procedure shall be non-returning if it overrides a dispatching non-returning procedure. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 6/2

Reason: This ensures that dispatching calls to non-returning procedures will, in fact, not return. 6.a/2

{*AI95-00414-01*} If a renaming-as-body completes a non-returning procedure declaration, then the renamed procedure shall be non-returning. 7/2

Reason: This ensures that no extra code is needed to implement the renames (that is, no wrapper is needed) as the body has the same property. 7.a/2

Static Semantics

{*AI95-00329-01*} {*AI95-00414-01*} If a generic procedure is non-returning, then so are its instances. If a procedure declared within a generic unit is non-returning, then so are the corresponding copies of that procedure in instances. 8/2

Dynamic Semantics

{*AI95-00329-01*} {*AI95-00414-01*} If the body of a non-returning procedure completes normally, Program_Error is raised at the point of the call. {*Program_Error (raised by failure of run-time check)*} 9/2

Discussion: Note that there is no name for suppressing this check, since the check represents a bug, imposes no time overhead, and minimal space overhead (since it can usually be statically eliminated as dead code). 9.a/2

Implementation Note: If a non-returning procedure tries to return, we raise Program_Error. This is stated as happening at the call site, because we do not wish to allow the procedure to handle the exception (and then, perhaps, try to return again!). However, the expected run-time model is that the compiler will generate **raise** Program_Error at the end of the procedure body (but not handleable by the procedure itself), as opposed to doing it at the call site. (This is just like the typical run-time model for functions that fall off the end without returning a value). The reason is indirect calls: in P.**all**(...);, the compiler cannot know whether P designates a non-returning procedure or a normal one. Putting the **raise** Program_Error in the procedure's generated code solves this problem neatly. 9.b/2

Similarly, if one passes a non-returning procedure to a generic formal parameter, the compiler cannot know this at call sites (in shared code implementations); the raise-in-body solution deals with this neatly. 9.c/2

Examples

10/2
```
{AI95-00433-01} procedure Fail(Msg : String);  -- raises Fatal_Error exception
pragma No_Return(Fail);
   -- Inform compiler and reader that procedure never returns normally
```

Extensions to Ada 95

{*AI95-00329-01*} {*AI95-00414-01*} {*extensions to Ada 95*} Pragma No_Return is new. 10.a/2

6.6 Overloading of Operators

{*operator*} {*user-defined operator*} {*operator (user-defined)*} An *operator* is a function whose designator is an operator_symbol. [Operators, like other functions, may be overloaded.] 1

Name Resolution Rules

Each use of a unary or binary operator is equivalent to a function_call with *function_*prefix being the corresponding operator_symbol, and with (respectively) one or two positional actual parameters being the operand(s) of the operator (in order). 2

2.a **To be honest:** We also use the term operator (in Section 4 and in 6.1) to refer to one of the syntactic categories defined in 4.5, “Operators and Expression Evaluation” whose names end with “_operator:” logical_operator, relational_-operator, binary_adding_operator, unary_adding_operator, multiplying_operator, and highest_precedence_-operator.

Legality Rules

3 The subprogram_specification of a unary or binary operator shall have one or two parameters, respectively. A generic function instantiation whose designator is an operator_symbol is only allowed if the specification of the generic function has the corresponding number of parameters.

4 Default_expressions are not allowed for the parameters of an operator (whether the operator is declared with an explicit subprogram_specification or by a generic_instantiation).

5 An explicit declaration of "/=" shall not have a result type of the predefined type Boolean.

Static Semantics

6 A declaration of "=" whose result type is Boolean implicitly declares a declaration of "/=" that gives the complementary result.

NOTES

7 8 The operators "+" and "–" are both unary and binary operators, and hence may be overloaded with both one- and two-parameter functions.

Examples

8 *Examples of user-defined operators:*

9
```
function "+" (Left, Right : Matrix) return Matrix;
function "+" (Left, Right : Vector) return Vector;

--  assuming that A, B, and C are of the type Vector
--  the following two statements are equivalent:

A := B + C;
A := "+"(B, C);
```

Extensions to Ada 83

9.a {*extensions to Ada 83*} Explicit declarations of "=" are now permitted for any combination of parameter and result types.

9.b Explicit declarations of "/=" are now permitted, so long as the result type is not Boolean.

6.7 Null Procedures

1/2 {*AI95-00348-01*} A null_procedure_declaration provides a shorthand to declare a procedure with an empty body.

Syntax

2/2 {*AI95-00348-01*} null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification **is null**;

Static Semantics

3/2 {*AI95-00348-01*} A null_procedure_declaration declares a *null procedure*.{*null procedure*} {*procedure (null)*} A completion is not allowed for a null_procedure_declaration.

3.a/2 **Reason:** There are no null functions because the return value has to be constructed somehow; a function that always raises Program_Error doesn't seem very useful or worth the complication.

Dynamic Semantics

{*AI95-00348-01*} The execution of a null procedure is invoked by a subprogram call. For the execution of a subprogram call on a null procedure, the execution of the subprogram_body has no effect. 4/2

Ramification: Thus, a null procedure is equivalent to the body 4.a/2

4.b/2
```
begin
   null;
end;
```

with the exception that a null procedure can be used in place of a procedure specification. 4.c/2

{*AI95-00348-01*} {*elaboration (null_procedure_declaration)* [partial]} The elaboration of a null_procedure_declaration has no effect. 5/2

Examples

6/2
```
{AI95-00433-01} procedure Simplify(Expr : in out Expression) is null; -- see 3.9
-- By default, Simplify does nothing, but it may be overridden in extensions of Expression
```

Extensions to Ada 95

{*AI95-00348-01*} {*extensions to Ada 95*} Null procedures are new. 6.a/2

Section 7: Packages

1 [{*Package*} [Glossary Entry]Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users. {*information hiding: See package*} {*encapsulation: See package*} {*module: See package*} {*class: See also package*}]

7.1 Package Specifications and Declarations

1 [A package is generally provided in two parts: a package_specification and a package_body. Every package has a package_specification, but not all packages have a package_body.]

Syntax

2 package_declaration ::= package_specification;

3 package_specification ::=
```
    package defining_program_unit_name is
      {basic_declarative_item}
   [private
      {basic_declarative_item}]
    end [[parent_unit_name.]identifier]
```

4 If an identifier or parent_unit_name.identifier appears at the end of a package_specification, then this sequence of lexical elements shall repeat the defining_program_unit_name.

Legality Rules

5/2 {*AI95-00434-01*} {*requires a completion (package_declaration)* [partial]} {*requires a completion (generic_package_declaration)* [partial]} A package_declaration or generic_package_declaration requires a completion [(a body)] if it contains any basic_declarative_item that requires a completion, but whose completion is not in its package_specification.

5.a **To be honest:** If an implementation supports it, a pragma Import may substitute for the body of a package or generic package.

Static Semantics

6/2 {*AI95-00420-01*} {*AI95-00434-01*} {*visible part (of a package (other than a generic formal package))* [partial]} The first list of basic_declarative_items of a package_specification of a package other than a generic formal package is called the *visible part* of the package. [{*private part (of a package)* [partial]} The optional list of basic_declarative_items after the reserved word **private** (of any package_specification) is called the *private part* of the package. If the reserved word **private** does not appear, the package has an implicit empty private part.] Each list of basic_declarative_items of a package_specification forms a *declaration list* of the package.{*declaration list (package_specification)* [partial]}

6.a **Ramification:** This definition of visible part does not apply to generic formal packages — 12.7 defines the visible part of a generic formal package.

6.b The implicit empty private part is important because certain implicit declarations occur there if the package is a child package, and it defines types in its visible part that are derived from, or contain as components, private types declared within the parent package. These implicit declarations are visible in children of the child package. See 10.1.1.

7 [An entity declared in the private part of a package is visible only within the declarative region of the package itself (including any child units — see 10.1.1). In contrast, expanded names denoting entities declared in the visible part can be used even outside the package; furthermore, direct visibility of such entities can be achieved by means of use_clauses (see 4.1.3 and 8.4).]

Dynamic Semantics

8 {*elaboration (package_declaration)* [partial]} The elaboration of a package_declaration consists of the elaboration of its basic_declarative_items in the given order.

NOTES

9 1 The visible part of a package contains all the information that another program unit is able to know about the package.

10 2 If a declaration occurs immediately within the specification of a package, and the declaration has a corresponding completion that is a body, then that body has to occur immediately within the body of the package.

10.a **Proof:** This follows from the fact that the declaration and completion are required to occur immediately within the same declarative region, and the fact that bodies are disallowed (by the Syntax Rules) in package_specifications. This does not apply to instances of generic units, whose bodies can occur in package_specifications.

Examples

11 *Example of a package declaration:*

12
```
package Rational_Numbers is
```
13
```
   type Rational is
      record
         Numerator   : Integer;
         Denominator : Positive;
      end record;
```
14
```
   function "="(X,Y : Rational) return Boolean;
```
15
```
   function "/"  (X,Y : Integer)  return Rational;  -- to construct a rational number
```
16
```
   function "+"  (X,Y : Rational) return Rational;
   function "-"  (X,Y : Rational) return Rational;
   function "*"  (X,Y : Rational) return Rational;
   function "/"  (X,Y : Rational) return Rational;
end Rational_Numbers;
```

17 There are also many examples of package declarations in the predefined language environment (see Annex A).

Incompatibilities With Ada 83

17.a {*incompatibilities with Ada 83*} In Ada 83, a library package is allowed to have a body even if it doesn't need one. In Ada 95, a library package body is either required or forbidden — never optional. The workaround is to add **pragma** Elaborate_Body, or something else requiring a body, to each library package that has a body that isn't otherwise required.

Wording Changes from Ada 83

17.b We have moved the syntax into this clause and the next clause from RM83-7.1, “Package Structure”, which we have removed.

17.c RM83 was unclear on the rules about when a package requires a body. For example, RM83-7.1(4) and RM83-7.1(8) clearly forgot about the case of an incomplete type declared in a package_declaration but completed in the body. In addition, RM83 forgot to make this rule apply to a generic package. We have corrected these rules. Finally, since we now allow a pragma Import for any explicit declaration, the completion rules need to take this into account as well.

Wording Changes from Ada 95

17.d/2 {*AI95-00420-01*} Defined “declaration list” to avoid ambiguity in other rules as to whether packages are included.

7.2 Package Bodies

1 [In contrast to the entities declared in the visible part of a package, the entities declared in the package_body are visible only within the package_body itself. As a consequence, a package with a package_body can be used for the construction of a group of related subprograms in which the logical operations available to clients are clearly isolated from the internal entities.]

Syntax

```
package_body ::=                                              2
   package body defining_program_unit_name is
     declarative_part
  [begin
     handled_sequence_of_statements]
   end [[parent_unit_name.]identifier];
```

If an identifier or parent_unit_name.identifier appears at the end of a package_body, then this sequence of lexical elements shall repeat the defining_program_unit_name. 3

Legality Rules

A package_body shall be the completion of a previous package_declaration or generic_package_-declaration. A library package_declaration or library generic_package_declaration shall not have a body unless it requires a body[; **pragma** Elaborate_Body can be used to require a library_unit_-declaration to have a body (see 10.2.1) if it would not otherwise require one]. 4

Ramification: The first part of the rule forbids a package_body from standing alone — it has to belong to some previous package_declaration or generic_package_declaration. 4.a

A nonlibrary package_declaration or nonlibrary generic_package_declaration that does not require a completion may have a corresponding body anyway. 4.b

Static Semantics

In any package_body without statements there is an implicit null_statement. For any package_-declaration without an explicit completion, there is an implicit package_body containing a single null_statement. For a noninstance, nonlibrary package, this body occurs at the end of the declarative_-part of the innermost enclosing program unit or block_statement; if there are several such packages, the order of the implicit package_bodies is unspecified. {*unspecified* [partial]} [(For an instance, the implicit package_body occurs at the place of the instantiation (see 12.3). For a library package, the place is partially determined by the elaboration dependences (see Section 10).)] 5

Discussion: Thus, for example, we can refer to something happening just after the **begin** of a package_body, and we can refer to the handled_sequence_of_statements of a package_body, without worrying about all the optional pieces. The place of the implicit body makes a difference for tasks activated by the package. See also RM83-9.3(5). 5.a

The implicit body would be illegal if explicit in the case of a library package that does not require (and therefore does not allow) a body. This is a bit strange, but not harmful. 5.b

Dynamic Semantics

{*elaboration (nongeneric package_body)* [partial]} For the elaboration of a nongeneric package_body, its declarative_part is first elaborated, and its handled_sequence_of_statements is then executed. 6

NOTES

3 A variable declared in the body of a package is only visible within this body and, consequently, its value can only be changed within the package_body. In the absence of local tasks, the value of such a variable remains unchanged between calls issued from outside the package to subprograms declared in the visible part. The properties of such a variable are similar to those of a “static” variable of C. 7

4 The elaboration of the body of a subprogram explicitly declared in the visible part of a package is caused by the elaboration of the body of the package. Hence a call of such a subprogram by an outside program unit raises the exception Program_Error if the call takes place before the elaboration of the package_body (see 3.11). 8

Examples

Example of a package body (see 7.1): 9

```
package body Rational_Numbers is                              10
```

11
```
      procedure Same_Denominator (X,Y : in out Rational) is
      begin
         --  reduces X and Y to the same denominator:
         ...
      end Same_Denominator;
```

12
```
      function "="(X,Y : Rational) return Boolean is
         U : Rational := X;
         V : Rational := Y;
      begin
         Same_Denominator (U,V);
         return U.Numerator = V.Numerator;
      end "=";
```

13
```
      function "/" (X,Y : Integer) return Rational is
      begin
         if Y > 0 then
            return (Numerator => X,  Denominator => Y);
         else
            return (Numerator => -X, Denominator => -Y);
         end if;
      end "/";
```

14
```
      function "+" (X,Y : Rational) return Rational is ... end "+";
      function "-" (X,Y : Rational) return Rational is ... end "-";
      function "*" (X,Y : Rational) return Rational is ... end "*";
      function "/" (X,Y : Rational) return Rational is ... end "/";
```

15
```
   end Rational_Numbers;
```

Wording Changes from Ada 83

15.a The syntax rule for package_body now uses the syntactic category handled_sequence_of_statements.

15.b The declarative_part of a package_body is now required; that doesn't make any real difference, since a declarative_part can be empty.

15.c RM83 seems to have forgotten to say that a package_body can't stand alone, without a previous declaration. We state that rule here.

15.d RM83 forgot to restrict the definition of elaboration of package_bodies to nongeneric ones. We have corrected that omission.

15.e The rule about implicit bodies (from RM83-9.3(5)) is moved here, since it is more generally applicable.

7.3 Private Types and Private Extensions

1 [The declaration (in the visible part of a package) of a type as a private type or private extension serves to separate the characteristics that can be used directly by outside program units (that is, the logical properties) from other characteristics whose direct use is confined to the package (the details of the definition of the type itself). See 3.9.1 for an overview of type extensions. {*private types and private extensions*} {*information hiding: See private types and private extensions*} {*opaque type: See private types and private extensions*} {*abstract data type (ADT): See private types and private extensions*} {*ADT (abstract data type): See private types and private extensions*}]

Language Design Principles

1.a A private (untagged) type can be thought of as a record type with the type of its single (hidden) component being the full view.

1.b A private tagged type can be thought of as a private extension of an anonymous parent with no components. The only dispatching operation of the parent is equality (although the Size attribute, and, if nonlimited, assignment are allowed, and those will presumably be implemented in terms of dispatching).

Syntax

2 private_type_declaration ::=
type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**;

{*AI95-00251-01*} {*AI95-00419-01*} {*AI95-00443-01*} private_extension_declaration ::= 3/2
  type defining_identifier [discriminant_part] **is**
   [**abstract**] [**limited** | **synchronized**] **new** *ancestor*_subtype_indication
   [**and** interface_list] **with private**;

Legality Rules

{*partial view (of a type)*} {*requires a completion (declaration of a partial view)* [partial]} A private_type_declaration or private_extension_declaration declares a *partial view* of the type; such a declaration is allowed only as a declarative_item of the visible part of a package, and it requires a completion, which shall be a full_type_declaration that occurs as a declarative_item of the private part of the package. [The view of the type declared by the full_type_declaration is called the *full view*.] A generic formal private type or a generic formal private extension is also a partial view. 4

To be honest: A private type can also be completed by a pragma Import, if supported by an implementation. 4.a

Reason: We originally used the term "private view," but this was easily confused with the view provided *from* the private part, namely the full view. 4.b

Proof: {*AI95-00326-01*} Full view is now defined in 3.2.1, "Type Declarations", as all types now have them. 4.c/2

[A type shall be completely defined before it is frozen (see 3.11.1 and 13.14). Thus, neither the declaration of a variable of a partial view of a type, nor the creation by an allocator of an object of the partial view are allowed before the full declaration of the type. Similarly, before the full declaration, the name of the partial view cannot be used in a generic_instantiation or in a representation item.] 5

Proof: This rule is stated officially in 3.11.1, "Completions of Declarations". 5.a

{*AI95-00419-01*} {*AI95-00443-01*} [A private type is limited if its declaration includes the reserved word **limited**; a private extension is limited if its ancestor type is a limited type that is not an interface type, or if the reserved word **limited** or **synchronized** appears in its definition.] If the partial view is nonlimited, then the full view shall be nonlimited. If a tagged partial view is limited, then the full view shall be limited. [On the other hand, if an untagged partial view is limited, the full view may be limited or nonlimited.] 6/2

If the partial view is tagged, then the full view shall be tagged. [On the other hand, if the partial view is untagged, then the full view may be tagged or untagged.] In the case where the partial view is untagged and the full view is tagged, no derivatives of the partial view are allowed within the immediate scope of the partial view; [derivatives of the full view are allowed.] 7

Ramification: Note that deriving from a partial view within its immediate scope can only occur in a package that is a child of the one where the partial view is declared. The rule implies that in the visible part of a public child package, it is impossible to derive from an untagged private type declared in the visible part of the parent package in the case where the full view of the parent type turns out to be tagged. We considered a model in which the derived type was implicitly redeclared at the earliest place within its immediate scope where characteristics needed to be added. However, we rejected that model, because (1) it would imply that (for an untagged type) subprograms explicitly declared after the derived type could be inherited, and (2) to make this model work for composite types as well, several implicit redeclarations would be needed, since new characteristics can become visible one by one; that seemed like too much mechanism. 7.a

Discussion: The rule for tagged partial views is redundant for partial views that are private extensions, since all extensions of a given ancestor tagged type are tagged, and limited if the ancestor is limited. We phrase this rule partially redundantly to keep its structure parallel with the other rules. 7.b

To be honest: This rule is checked in a generic unit, rather than using the "assume the best" or "assume the worst" method. 7.c

Reason: {*AI95-00230-01*} Tagged limited private types have certain capabilities that are incompatible with having assignment for the full view of the type. In particular, tagged limited private types can be extended with components of a limited type, which works only because assignment is not allowed. Consider the following example: 7.d/2

7.e
```
package P1 is
    type T1 is tagged limited private;
    procedure Foo(X : in T1'Class);
private
    type T1 is tagged null record; -- Illegal!
        -- This should say “tagged limited null record”.
end P1;
```

7.f/1
```
package body P1 is
    type A is access T1'Class;
    Global : A;
    procedure Foo(X : in T1'Class) is
    begin
        Global := new T1'Class'(X);
            -- This would be illegal if the full view of
            -- T1 were limited, like it's supposed to be.
    end Foo;
end P1;
```

7.g/2
```
{AI95-00230-01} with P1;
package P2 is
    type T2(D : access Integer)
            is new P1.T1 with
        record
            My_Task : Some_Task_Type; -- Trouble!
        end record;
end P2;
```

7.h/1
```
with P1;
with P2;
procedure Main is
    Local : aliased Integer;
    Y : P2.T2(D => Local'Access);
begin
    P1.Foo(Y);
end Main;
```

7.i/2 {*AI95-00230-01*} If the above example were legal, we would have succeeded in doing an assignment of a task object, which is supposed to be a no-no.

7.j This rule is not needed for private extensions, because they inherit their limitedness from their ancestor, and there is a separate rule forbidding limited components of the corresponding record extension if the parent is nonlimited.

7.k **Ramification:** A type derived from an untagged private type is untagged, even if the full view of the parent is tagged, and even at places that can see the parent:

7.l
```
package P is
    type Parent is private;
private
    type Parent is tagged
        record
            X: Integer;
        end record;
end P;
```

7.m/1
```
with P;
package Q is
    type T is new P.Parent;
end Q;
```

7.n
```
with Q; use Q;
package body P is
    ... T'Class ... -- Illegal!
    Object: T;
    ... Object.X ... -- Illegal!
    ... Parent(Object).X ... -- OK.
end P;
```

7.o The declaration of T declares an untagged view. This view is always untagged, so T'Class is illegal, it would be illegal to extend T, and so forth. The component name X is never visible for this view, although the component is still there — one can get one's hands on it via a type_conversion.

7.1/2 {*AI95-00396-01*} If a full type has a partial view that is tagged, then:

7.2/2
- the partial view shall be a synchronized tagged type (see 3.9.4) if and only if the full type is a synchronized tagged type;

Reason: Since we do not allow record extensions of synchronized tagged types, this property has to be visible in the partial view to avoid privacy breaking. Generic formals do not need a similar rule as any extensions are rechecked for legality in the specification, and extensions of tagged formals are always illegal in a generic body. 7.o.1/2

- the partial view shall be a descendant of an interface type (see 3.9.4) if and only if the full type is a descendant of the interface type. 7.3/2

Reason: Consider the following example: 7.p/2

```
package P is                                                              7.q/2
   package Pkg is
      type Ifc is interface;
      procedure Foo (X : Ifc) is abstract;
   end Pkg;

   type Parent_1 is tagged null record;                                   7.r/2

   type T1 is new Parent_1 with private;                                  7.s/2
private
   type Parent_2 is new Parent_1 and Pkg.Ifc with null record;
   procedure Foo (X : Parent_2); -- Foo #1

   type T1 is new Parent_2 with null record; -- Illegal.                  7.t/2
end P;

with P;                                                                   7.u/2
package P_Client is
   type T2 is new P.T1 and P.Pkg.Ifc with null record;
   procedure Foo (X : T2); -- Foo #2
   X : T2;
end P_Client;

with P_Client;                                                            7.v/2
package body P is
   ...

   procedure Bar (X : T1'Class) is                                        7.w/2
   begin
      Pkg.Foo (X); -- should call Foo #1 or an override thereof
   end;

begin                                                                     7.x/2
   Pkg.Foo (Pkg.Ifc'Class (P_Client.X));      -- should call Foo #2
   Bar (T1'Class (P_Client.X));
end P;
```

This example is illegal because the completion of T1 is descended from an interface that the partial view is not descended from. If it were legal, T2 would implement Ifc twice, once in the visible part of P, and once in the visible part of P_Client. We would need to decide how Foo #1 and Foo #2 relate to each other. There are two options: either Foo #2 overrides Foo #1, or it doesn't. 7.y/2

If Foo #2 overrides Foo #1, we have a problem because the client redefines a behavior that it doesn't know about, and we try to avoid this at all costs, as it would lead to a breakdown of whatever abstraction was implemented. If the abstraction didn't expose that it implements Ifc, there must be a reason, and it should be able to depend on the fact that no overriding takes place in clients. Also, during maintenance, things may change and the full view might implement a different set of interfaces. Furthermore, the situation is even worse if the full type implements another interface Ifc2 that happens to have a conforming Foo (otherwise unrelated, except for its name and profile). 7.z/2

If Foo #2 doesn't override Foo #1, there is some similarity with the case of normal tagged private types, where a client can declare an operation that happens to conform to some private operation, and that's OK, it gets a different slot in the type descriptor. The problem here is that T2 would implement Ifc in two different ways, and through conversions to Ifc'Class we could end up with visibility on both of these two different implementations. This is the “diamond inheritance” problem of C++ all over again, and we would need some kind of a preference rule to pick one implementation. We don't want to go there (if we did, we might as well provide full-fledged multiple inheritance). 7.aa/2

Note that there wouldn't be any difficulty to implement the first option, so the restriction is essentially methodological. The second option might be harder to implement, depending on the language rules that we would choose. 7.bb/2

Ramification: This rule also prevents completing a private type with an interface. A interface, like all types, is a descendant of itself, and thus this rule is triggered. One reason this is necessary is that a client of a private extension should be able to inherit limitedness without having to look in the private part to see if the type is an interface (remember that limitedness of interfaces is never inherited, while it is inherited from other types). 7.cc/2

{*ancestor subtype (of a private_extension_declaration)*} The *ancestor subtype* of a private_extension_declaration is the subtype defined by the *ancestor*_subtype_indication; the ancestor type shall be a specific tagged type. The full view of a private extension shall be derived (directly or 8 indirectly) from the ancestor type. In addition to the places where Legality Rules normally apply (see 12.3), the requirement that the ancestor be specific applies also in the private part of an instance of a generic unit.

8.a **Reason:** This rule allows the full view to be defined through several intermediate derivations, possibly from a series of types produced by generic_instantiations.

8.1/2 {*AI95-00419-01*} {*AI95-00443-01*} If the reserved word **limited** appears in a private_extension_declaration, the ancestor type shall be a limited type. If the reserved word **synchronized** appears in a private_extension_declaration, the ancestor type shall be a limited interface.

9 If the declaration of a partial view includes a known_discriminant_part, then the full_type_declaration shall have a fully conforming [(explicit)] known_discriminant_part [(see 6.3.1, “Conformance Rules”)]. {*full conformance (required)*} [The ancestor subtype may be unconstrained; the parent subtype of the full view is required to be constrained (see 3.7).]

9.a **Discussion:** If the ancestor subtype has discriminants, then it is usually best to make it unconstrained.

9.b **Ramification:** If the partial view has a known_discriminant_part, then the full view has to be a composite, non-array type, since only such types may have known discriminants. Also, the full view cannot inherit the discriminants in this case; the known_discriminant_part has to be explicit.

9.c That is, the following is illegal:

9.d
```
package P is
    type T(D : Integer) is private;
private
    type T is new Some_Other_Type; -- Illegal!
end P;
```

9.e even if Some_Other_Type has an integer discriminant called D.

9.f It is a ramification of this and other rules that in order for a tagged type to privately inherit unconstrained discriminants, the private type declaration has to have an unknown_discriminant_part.

10 If a private extension inherits known discriminants from the ancestor subtype, then the full view shall also inherit its discriminants from the ancestor subtype, and the parent subtype of the full view shall be constrained if and only if the ancestor subtype is constrained.

10.a **Reason:** The first part ensures that the full view has the same discriminants as the partial view. The second part ensures that if the partial view is unconstrained, then the full view is also unconstrained; otherwise, a client might constrain the partial view in a way that conflicts with the constraint on the full view.

10.1/2 {*AI95-00419-01*} If the full_type_declaration for a private extension is defined by a derived_type_definition, then the reserved word **limited** shall appear in the full_type_declaration if and only if it also appears in the private_extension_declaration.

10.b/2 **Reason:** The word **limited** is optional (unless the ancestor is an interface), but it should be used consistently. Otherwise things would be too confusing for the reader. Of course, we only require that if the full type is defined by a derived_type_definition, as we want to allow task and protected types to complete extensions of synchronized interfaces.

11 [If a partial view has unknown discriminants, then the full_type_declaration may define a definite or an indefinite subtype, with or without discriminants.]

12 If a partial view has neither known nor unknown discriminants, then the full_type_declaration shall define a definite subtype.

13 If the ancestor subtype of a private extension has constrained discriminants, then the parent subtype of the full view shall impose a statically matching constraint on those discriminants. {*statically matching (required)* [partial]}

13.a **Ramification:** If the parent type of the full view is not the ancestor type, but is rather some descendant thereof, the constraint on the discriminants of the parent type might come from the declaration of some intermediate type in the derivation chain between the ancestor type and the parent type.

Reason: This prevents the following: 13.b

```
package P is                                                    13.c
    type T2 is new T1(Discrim => 3) with private;
private
    type T2 is new T1(Discrim => 999) -- Illegal!
        with record ...;
end P;
```

The constraints in this example do not statically match. 13.d

If the constraint on the parent subtype of the full view depends on discriminants of the full view, then the ancestor subtype has to be unconstrained: 13.e

```
type One_Discrim(A: Integer) is tagged ...;                     13.f
...
package P is
    type Two_Discrims(B: Boolean; C: Integer) is new One_Discrim with private;
private
    type Two_Discrims(B: Boolean; C: Integer) is new One_Discrim(A => C) with
        record
            ...
        end record;
end P;
```

The above example would be illegal if the private extension said “is new One_Discrim(A => C);”, because then the constraints would not statically match. (Constraints that depend on discriminants are not static.) 13.g

Static Semantics

14 {*private type* [partial]} A private_type_declaration declares a private type and its first subtype. {*private extension* [partial]} Similarly, a private_extension_declaration declares a private extension and its first subtype.

14.a **Discussion:** {*package-private type*} A *package-private type* is one declared by a private_type_declaration; that is, a private type other than a generic formal private type. {*package-private extension*} Similarly, a *package-private extension* is one declared by a private_extension_declaration. These terms are not used in the RM95 version of this document.

15 A declaration of a partial view and the corresponding full_type_declaration define two views of a single type. The declaration of a partial view together with the visible part define the operations that are available to outside program units; the declaration of the full view together with the private part define other operations whose direct use is possible only within the declarative region of the package itself. {*characteristics*} Moreover, within the scope of the declaration of the full view, the *characteristics* of the type are determined by the full view; in particular, within its scope, the full view determines the classes that include the type, which components, entries, and protected subprograms are visible, what attributes and other predefined operations are allowed, and whether the first subtype is static. See 7.3.1.

16/2 {*AI95-00401*} A private extension inherits components (including discriminants unless there is a new discriminant_part specified) and user-defined primitive subprograms from its ancestor type and its progenitor types (if any), in the same way that a record extension inherits components and user-defined primitive subprograms from its parent type and its progenitor types (see 3.4).

16.a **To be honest:** If an operation of the parent type is abstract, then the abstractness of the inherited operation is different for nonabstract record extensions than for nonabstract private extensions (see 3.9.3).

Dynamic Semantics

17 {*elaboration (private_type_declaration)* [partial]} The elaboration of a private_type_declaration creates a partial view of a type. {*elaboration (private_extension_declaration)* [partial]} The elaboration of a private_extension_declaration elaborates the *ancestor*_subtype_indication, and creates a partial view of a type.

NOTES

18 5 The partial view of a type as declared by a private_type_declaration is defined to be a composite view (in 3.2). The full view of the type might or might not be composite. A private extension is also composite, as is its full view.

19/2 6 {*AI95-00318-02*} Declaring a private type with an unknown_discriminant_part is a way of preventing clients from creating uninitialized objects of the type; they are then forced to initialize each object by calling some operation declared in the visible part of the package.

19.a **Discussion:** {*generic contract/private type contract analogy*} Packages with private types are analogous to generic packages with formal private types, as follows: The declaration of a package-private type is like the declaration of a formal private type. The visible part of the package is like the generic formal part; these both specify a contract (that is, a set of operations and other things available for the private type). The private part of the package is like an instantiation of the generic; they both give a full_type_declaration that specifies implementation details of the private type. The clients of the package are like the body of the generic; usage of the private type in these places is restricted to the operations defined by the contract.

19.b In other words, being inside the package is like being outside the generic, and being outside the package is like being inside the generic; a generic is like an "inside-out" package.

19.c This analogy also works for private extensions in the same inside-out way.

19.d Many of the legality rules are defined with this analogy in mind. See, for example, the rules relating to operations of [formal] derived types.

19.e The completion rules for a private type are intentionally quite similar to the matching rules for a generic formal private type.

19.f This analogy breaks down in one respect: a generic actual subtype is a subtype, whereas the full view for a private type is always a new type. (We considered allowing the completion of a private_type_declaration to be a subtype_declaration, but the semantics just won't work.) This difference is behind the fact that a generic actual type can be class-wide, whereas the completion of a private type always declares a specific type.

20/2 7 {*AI95-00401*} The ancestor type specified in a private_extension_declaration and the parent type specified in the corresponding declaration of a record extension given in the private part need not be the same. If the ancestor type is not an interface type, the parent type of the full view can be any descendant of the ancestor type. In this case, for a primitive subprogram that is inherited from the ancestor type and not overridden, the formal parameter names and default expressions (if any) come from the corresponding primitive subprogram of the specified ancestor type, while the body comes from the corresponding primitive subprogram of the parent type of the full view. See 3.9.2.

20.1/2 8 {*AI95-00401*} If the ancestor type specified in a private_extension_declaration is an interface type, the parent type can be any type so long as the full view is a descendant of the ancestor type. The progenitor types specified in a private_extension_declaration and the progenitor types specified in the corresponding declaration of a record extension given in the private part need not be the same — the only requirement is that the private extension and the record extension be descended from the same set of interfaces.

Examples

21 *Examples of private type declarations:*

22
```
type Key is private;
type File_Name is limited private;
```

23 *Example of a private extension declaration:*

24
```
type List is new Ada.Finalization.Controlled with private;
```

Extensions to Ada 83

24.a {*extensions to Ada 83*} The syntax for a private_type_declaration is augmented to allow the reserved word **tagged**.

24.b In Ada 83, a private type without discriminants cannot be completed with a type with discriminants. Ada 95 allows the full view to have discriminants, so long as they have defaults (that is, so long as the first subtype is definite). This change is made for uniformity with generics, and because the rule as stated is simpler and easier to remember than the Ada 83 rule. In the original version of Ada 83, the same restriction applied to generic formal private types. However, the restriction was removed by the ARG for generics. In order to maintain the "generic contract/private type contract analogy" discussed above, we have to apply the same rule to package-private types. Note that a private untagged type without discriminants can be completed with a tagged type with discriminants only if the full view is constrained, because discriminants of tagged types cannot have defaults.

Wording Changes from Ada 83

24.c RM83-7.4.1(4), "Within the specification of the package that declares a private type and before the end of the corresponding full type declaration, a restriction applies....", is subsumed (and corrected) by the rule that a type shall be completely defined before it is frozen, and the rule that the parent type of a derived type declaration shall be completely defined, unless the derived type is a private extension.

Extensions to Ada 95

24.d/2 {*AI95-00251-01*} {*AI95-00396-01*} {*AI95-00401-01*} {*extensions to Ada 95*} Added interface_list to private extensions to support interfaces and multiple inheritance (see 3.9.4).

24.e/2 {*AI95-00419-01*} A private extension may specify that it is a limited type. This is required for interface ancestors (from which limitedness is not inherited), but it is generally useful as documentation of limitedness.

24.f/2 {*AI95-00443-01*} A private extension may specify that it is a synchronized type. This is required in order so that a regular limited interface can be used as the ancestor of a synchronized type (we do not allow hiding of synchronization).

7.3.1 Private Operations

1 [For a type declared in the visible part of a package or generic package, certain operations on the type do not become visible until later in the package — either in the private part or the body. {*private operations*} Such *private operations* are available only inside the declarative region of the package or generic package.]

Static Semantics

2 The predefined operators that exist for a given type are determined by the classes to which the type belongs. For example, an integer type has a predefined "+" operator. In most cases, the predefined operators of a type are declared immediately after the definition of the type; the exceptions are explained below. Inherited subprograms are also implicitly declared immediately after the definition of the type, except as stated below.

3/1 {*8652/0019*} {*AI95-00033-01*} For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later immediately within the declarative region in which the composite type is declared additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place.

4/1 {*8652/0019*} {*AI95-00033-01*} The corresponding rule applies to a type defined by a derived_type_definition, if there is a place immediately within the declarative region in which the type is declared where additional characteristics of its parent type become visible.

5/1 {*8652/0019*} {*AI95-00033-01*} {*become nonlimited*} {*nonlimited type (becoming nonlimited)*} {*limited type (becoming nonlimited)*} [For example, an array type whose component type is limited private becomes nonlimited if the full view of the component type is nonlimited and visible at some later place immediately within the declarative region in which the array type is declared. In such a case, the predefined "=" operator is implicitly declared at that place, and assignment is allowed after that place.]

6/1 {*8652/0019*} {*AI95-00033-01*} Inherited primitive subprograms follow a different rule. For a derived_type_definition, each inherited primitive subprogram is implicitly declared at the earliest place, if any, immediately within the declarative region in which the type_declaration occurs, but after the type_declaration, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all. [An inherited subprogram that is not declared at all cannot be named in a call and cannot be overridden, but for a tagged type, it is possible to dispatch to it.]

7 For a private_extension_declaration, each inherited subprogram is declared immediately after the private_extension_declaration if the corresponding declaration from the ancestor is visible at that place. Otherwise, the inherited subprogram is not declared for the private extension, [though it might be for the full type].

7.a/1 **Reason:** There is no need for the “earliest place immediately within the declarative region” business here, because a private_extension_declaration will be completed with a full_type_declaration, so we can hang the necessary private implicit declarations on the full_type_declaration.

7.b **Discussion:** The above rules matter only when the component type (or parent type) is declared in the visible part of a package, and the composite type (or derived type) is declared within the declarative region of that package (possibly in a nested package or a child package).

7.c Consider:

7.d
```
package Parent is
    type Root is tagged null record;
    procedure Op1(X : Root);
```

7.e
```
    type My_Int is range 1..10;
private
    procedure Op2(X : Root);
```

7.f
```
    type Another_Int is new My_Int;
    procedure Int_Op(X : My_Int);
end Parent;
```

7.g
```
with Parent; use Parent;
package Unrelated is
    type T2 is new Root with null record;
    procedure Op2(X : T2);
end Unrelated;
```

7.h
```
package Parent.Child is
    type T3 is new Root with null record;
    -- Op1(T3) implicitly declared here.
```

7.i
```
    package Nested is
        type T4 is new Root with null record;
    private
        ...
    end Nested;
private
    -- Op2(T3) implicitly declared here.
    ...
end Parent.Child;
```

7.j
```
with Unrelated; use Unrelated;
package body Parent.Child is
    package body Nested is
        -- Op2(T4) implicitly declared here.
    end Nested;
```

7.k
```
    type T5 is new T2 with null record;
end Parent.Child;
```

7.l Another_Int does not inherit Int_Op, because Int_Op does not “exist” at the place where Another_Int is declared.

7.m/1 Type T2 inherits Op1 and Op2 from Root. However, the inherited Op2 is never declared, because Parent.Op2 is never visible immediately within the declarative region of T2. T2 explicitly declares its own Op2, but this is unrelated to the inherited one — it does not override the inherited one, and occupies a different slot in the type descriptor.

7.n T3 inherits both Op1 and Op2. Op1 is implicitly declared immediately after the type declaration, whereas Op2 is declared at the beginning of the private part. Note that if Child were a private child of Parent, then Op1 and Op2 would both be implicitly declared immediately after the type declaration.

7.o/1 T4 is similar to T3, except that the earliest place immediately within the declarative region containing T4 where Root's Op2 is visible is in the body of Nested.

7.p If T3 or T4 were to declare a type-conformant Op2, this would override the one inherited from Root. This is different from the situation with T2.

7.q T5 inherits Op1 and two Op2's from T2. Op1 is implicitly declared immediately after the declaration of T5, as is the Op2 that came from Unrelated.Op2. However, the Op2 that originally came from Parent.Op2 is never implicitly declared for T5, since T2's version of that Op2 is never visible (anywhere — it never got declared either).

7.r For all of these rules, implicit private parts and bodies are assumed as needed.

7.s It is possible for characteristics of a type to be revealed in more than one place:

7.t
```
package P is
    type Comp1 is private;
private
    type Comp1 is new Boolean;
end P;
```

7.u
```
package P.Q is
    package R is
        type Comp2 is limited private;
        type A is array(Integer range <>) of Comp2;
    private
        type Comp2 is new Comp1;
        -- A becomes nonlimited here.
        -- "="(A, A) return Boolean is implicitly declared here.
        ...
    end R;
private
    -- Now we find out what Comp1 really is, which reveals
    -- more information about Comp2, but we're not within
    -- the immediate scope of Comp2, so we don't do anything
    -- about it yet.
end P.Q;
```

7.v
```
package body P.Q is
    package body R is
        -- Things like "xor"(A,A) return A are implicitly
        -- declared here.
    end R;
end P.Q;
```

7.v.1/1 {*8652/0019*} {*AI95-00033-01*} We say *immediately* within the declarative region in order that types do not gain operations within a nested scope. Consider:

7.v.2/1
```
package Outer is
    package Inner is
        type Inner_Type is private;
    private
        type Inner_Type is new Boolean;
    end Inner;
    type Outer_Type is array(Natural range <>) of Inner.Inner_Type;
end Outer;
```

7.v.3/1
```
package body Outer is
    package body Inner is
        -- At this point, we can see that Inner_Type is a Boolean type.
        -- But we don't want Outer_Type to gain an "and" operator here.
    end Inner;
end Outer;
```

8 [The Class attribute is defined for tagged subtypes in 3.9. In addition,] for every subtype S of an untagged private type whose full view is tagged, the following attribute is defined:

9 S'Class Denotes the class-wide subtype corresponding to the full view of S. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. [After the full view, the Class attribute of the full view can be used.]

NOTES

10 9 Because a partial view and a full view are two different views of one and the same type, outside of the defining package the characteristics of the type are those defined by the visible part. Within these outside program units the type is just a private type or private extension, and any language rule that applies only to another class of types does not apply. The fact that the full declaration might implement a private type with a type of a particular class (for example, as an array type) is relevant only within the declarative region of the package itself including any child units.

11 The consequences of this actual implementation are, however, valid everywhere. For example: any default initialization of components takes place; the attribute Size provides the size of the full view; finalization is still done for controlled components of the full view; task dependence rules still apply to components that are task objects.

12/2 10 {*AI95-00287-01*} Partial views provide initialization, membership tests, selected components for the selection of discriminants and inherited components, qualification, and explicit conversion. Nonlimited partial views also allow use of assignment_statements.

13 11 For a subtype S of a partial view, S'Size is defined (see 13.3). For an object A of a partial view, the attributes A'Size and A'Address are defined (see 13.3). The Position, First_Bit, and Last_Bit attributes are also defined for discriminants and inherited components.

Examples

14 *Example of a type with private operations:*

15
```
package Key_Manager is
   type Key is private;
   Null_Key : constant Key; -- a deferred constant declaration (see 7.4)
   procedure Get_Key(K : out Key);
   function "<" (X, Y : Key) return Boolean;
private
   type Key is new Natural;
   Null_Key : constant Key := Key'First;
end Key_Manager;
```

16
```
package body Key_Manager is
   Last_Key : Key := Null_Key;
   procedure Get_Key(K : out Key) is
   begin
      Last_Key := Last_Key + 1;
      K := Last_Key;
   end Get_Key;
```

17
```
   function "<" (X, Y : Key) return Boolean is
   begin
      return Natural(X) < Natural(Y);
   end "<";
end Key_Manager;
```

NOTES

18 12 *Notes on the example:* Outside of the package Key_Manager, the operations available for objects of type Key include assignment, the comparison for equality or inequality, the procedure Get_Key and the operator "<"; they do not include other relational operators such as ">=", or arithmetic operators.

19 The explicitly declared operator "<" hides the predefined operator "<" implicitly declared by the full_type_declaration. Within the body of the function, an explicit conversion of X and Y to the subtype Natural is necessary to invoke the "<" operator of the parent type. Alternatively, the result of the function could be written as not (X >= Y), since the operator ">=" is not redefined.

20 The value of the variable Last_Key, declared in the package body, remains unchanged between calls of the procedure Get_Key. (See also the NOTES of 7.2.)

Wording Changes from Ada 83

20.a The phrase in RM83-7.4.2(7), "...after the full type declaration", doesn't work in the presence of child units, so we define that rule in terms of visibility.

20.b The definition of the Constrained attribute for private types has been moved to "Obsolescent Features." (The Constrained attribute of an object has not been moved there.)

Wording Changes from Ada 95

20.c/2 {*8652/0018*} {*AI95-00033-01*} **Corrigendum:** Clarified when additional operations are declared.

20.d/2 {*AI95-00287-01*} Revised the note on operations of partial views to reflect that limited types do have an assignment operation, but not assignment_statements.

7.4 Deferred Constants

1 [Deferred constant declarations may be used to declare constants in the visible part of a package, but with the value of the constant given in the private part. They may also be used to declare constants imported from other languages (see Annex B).]

Legality Rules

2 [{*deferred constant declaration*} A *deferred constant declaration* is an object_declaration with the reserved word **constant** but no initialization expression.] {*deferred constant*} The constant declared by a deferred constant declaration is called a *deferred constant*. {*requires a completion (deferred constant declaration)*

[partial]} A deferred constant declaration requires a completion, which shall be a full constant declaration (called the *full declaration* of the deferred constant), or a pragma Import (see Annex B). {*full declaration*}

Proof: The first sentence is redundant, as it is stated officially in 3.3.1. 2.a

3 A deferred constant declaration that is completed by a full constant declaration shall occur immediately within the visible part of a package_specification. For this case, the following additional rules apply to the corresponding full declaration:

- 4 The full declaration shall occur immediately within the private part of the same package;
- 5/2 {*AI95-00385-01*} The deferred and full constants shall have the same type, or shall have statically matching anonymous access subtypes;

  5.a/2 **Ramification:** {*AI95-00385-01*} This implies that both the deferred declaration and the full declaration have to have a subtype_indication or access_definition rather than an array_type_definition, because each array_type_definition would define a new type.

- 6/2 {*AI95-00385-01*} If the deferred constant declaration includes a subtype_indication that defines a constrained subtype, then the subtype defined by the subtype_indication in the full declaration shall match it statically.[On the other hand, if the subtype of the deferred constant is unconstrained, then the full declaration is still allowed to impose a constraint. The constant itself will be constrained, like all constants;]
- 7/2 {*AI95-00231-01*} If the deferred constant declaration includes the reserved word **aliased**, then the full declaration shall also;

  7.a **Ramification:** On the other hand, the full constant can be aliased even if the deferred constant is not.

- 7.1/2 {*AI95-00231-01*} If the subtype of the deferred constant declaration excludes null, the subtype of the full declaration shall also exclude null.

  7.a.1/2 **Ramification:** On the other hand, the full constant can exclude null even if the deferred constant does not. But that can only happen for a subtype_indication, as anonymous access types are required to statically match (which includes any null_exclusion).

8 [A deferred constant declaration that is completed by a pragma Import need not appear in the visible part of a package_specification, and has no full constant declaration.]

9/2 {*AI95-00256-01*} The completion of a deferred constant declaration shall occur before the constant is frozen (see 13.14).

Dynamic Semantics

10 {*elaboration (deferred constant declaration)* [partial]} The elaboration of a deferred constant declaration elaborates the subtype_indication or (only allowed in the case of an imported constant) the array_type_definition.

NOTES

11 13 The full constant declaration for a deferred constant that is of a given private type or private extension is not allowed before the corresponding full_type_declaration. This is a consequence of the freezing rules for types (see 13.14).

11.a **Ramification:** Multiple or single declarations are allowed for the deferred and the full declarations, provided that the equivalent single declarations would be allowed.

11.b Deferred constant declarations are useful for declaring constants of private views, and types with components of private views. They are also useful for declaring access-to-constant objects that designate variables declared in the private part of a package.

Examples

12 *Examples of deferred constant declarations:*

13
```
Null_Key : constant Key;      -- see 7.3.1
```

14
```
    CPU_Identifier : constant String(1..8);
    pragma Import(Assembler, CPU_Identifier, Link_Name => "CPU_ID");
                                            -- see B.1
```

Extensions to Ada 83

14.a {*extensions to Ada 83*} In Ada 83, a deferred constant is required to be of a private type declared in the same visible part. This restriction is removed for Ada 95; deferred constants can be of any type.

14.b In Ada 83, a deferred constant declaration was not permitted to include a constraint, nor the reserved word **aliased**.

14.c In Ada 83, the rules required conformance of type marks; here we require static matching of subtypes if the deferred constant is constrained.

14.d A deferred constant declaration can be completed with a pragma Import. Such a deferred constant declaration need not be within a package_specification.

14.e The rules for too-early uses of deferred constants are modified in Ada 95 to allow more cases, and catch all errors at compile time. This change is necessary in order to allow deferred constants of a tagged type without violating the principle that for a dispatching call, there is always an implementation to dispatch to. It has the beneficial side-effect of catching some Ada-83-erroneous programs at compile time. The new rule fits in well with the new freezing-point rules. Furthermore, we are trying to convert undefined-value problems into bounded errors, and we were having trouble for the case of deferred constants. Furthermore, uninitialized deferred constants cause trouble for the shared variable / tasking rules, since they are really variable, even though they purport to be constant. In Ada 95, they cannot be touched until they become constant.

14.f Note that we do not consider this change to be an upward incompatibility, because it merely changes an erroneous execution in Ada 83 into a compile-time error.

14.g The Ada 83 semantics are unclear in the case where the full view turns out to be an access type. It is a goal of the language design to prevent uninitialized access objects. One wonders if the implementation is required to initialize the deferred constant to null, and then initialize it (again!) to its real value. In Ada 95, the problem goes away.

Wording Changes from Ada 83

14.h Since deferred constants can now be of a nonprivate type, we have made this a stand-alone clause, rather than a subclause of 7.3, “Private Types and Private Extensions”.

14.i Deferred constant declarations used to have their own syntax, but now they are simply a special case of object_declarations.

Extensions to Ada 95

14.j/2 {*AI95-00385-01*} {*extensions to Ada 95*} Deferred constants were enhanced to allow the use of anonymous access types in them.

Wording Changes from Ada 95

14.k/2 {*AI95-00231-01*} Added matching rules for subtypes that exclude null.

7.5 Limited Types

1/2 {*AI95-00287-01*} [A limited type is (a view of) a type for which copying (such as for an assignment_statement) is not allowed. A nonlimited type is a (view of a) type for which copying is allowed.]

1.a **Discussion:** The concept of the *value* of a limited type is difficult to define, since the abstract value of a limited type often extends beyond its physical representation. In some sense, values of a limited type cannot be divorced from their object. The value *is* the object.

1.b/2 {*AI95-00318-02*} In Ada 83, in the two places where limited types were defined by the language, namely tasks and files, an implicit level of indirection was implied by the semantics to avoid the separation of the value from an associated object. In Ada 95, most limited types are passed by reference, and even return-ed by reference. In Ada 2005, most limited types are built-in-place upon return, rather than returned by reference. Thus the object “identity” is part of the logical value of most limited types.

1.c/2 **To be honest:** {*AI95-00287-01*} {*AI95-00419-01*} For a limited partial view whose full view is nonlimited, copying is possible on parameter passing and function return. To prevent any copying whatsoever, one should make both the partial *and* full views limited.

Glossary entry: {*Limited type*} A limited type is a type for which copying (such as in an assignment_statement) is not allowed. A nonlimited type is a type for which copying is allowed. 1.d/2

Legality Rules

{*AI95-00419-01*} If a tagged record type has any limited components, then the reserved word **limited** shall appear in its record_type_definition. [If the reserved word **limited** appears in the definition of a derived_type_definition, its parent type and any progenitor interfaces shall be limited.] 2/2

Proof: {*AI95-00419-01*} The rule about the parent type being required to be limited can be found in 3.4. Rules about progenitor interfaces can be found in 3.9.4, specifically, a nonlimited interface can appear only on a nonlimited type. We repeat these rules here to gather these scattered rules in one obvious place. 2.a.1/2

Reason: This prevents tagged limited types from becoming nonlimited. Otherwise, the following could happen: 2.a

2.b
```
package P is
    type T is limited private;
    type R is tagged
        record -- Illegal!
               -- This should say "limited record".
            X : T;
        end record;
private
    type T is new Integer; -- R becomes nonlimited here.
end P;
```

2.c/2
```
package Q is
    type R2 is new R with
        record
            Y : Some_Task_Type;
        end record;
end Q;
```

{*AI95-00230-01*} If the above were legal, then assignment would be defined for R'Class in the body of P, which is bad news, given the task. 2.d/2

{*AI95-00287-01*} {*AI95-00318-02*} In the following contexts, an expression of a limited type is not permitted unless it is an aggregate, a function_call, or a parenthesized expression or qualified_expression whose operand is permitted by this rule: 2.1/2

- the initialization expression of an object_declaration (see 3.3.1) 2.2/2
- the default_expression of a component_declaration (see 3.8) 2.3/2
- the expression of a record_component_association (see 4.3.1) 2.4/2
- the expression for an ancestor_part of an extension_aggregate (see 4.3.2) 2.5/2
- an expression of a positional_array_aggregate or the expression of an array_component_association (see 4.3.3) 2.6/2
- the qualified_expression of an initialized allocator (see 4.8) 2.7/2
- the expression of a return statement (see 6.5) 2.8/2
- the default_expression or actual parameter for a formal object of mode **in** (see 12.4) 2.9/2

Discussion: All of these contexts normally require copying; by restricting the uses as above, we can require the new object to be built-in-place. 2.e/2

Static Semantics

{*AI95-00419-01*} {*limited type*} A type is *limited* if it is one of the following: 3/2

- {*AI95-00411-01*} {*AI95-00419-01*} a type with the reserved word **limited**, **synchronized**, **task**, or **protected** in its definition; 4/2

Ramification: Note that there is always a "definition," conceptually, even if there is no syntactic category called "..._definition". 4.a

{*AI95-00419-01*} This includes interfaces of the above kinds, derived types with the reserved word **limited**, as well as task and protected types. 4.b/2

5/2 • *This paragraph was deleted.*{*AI95-00419-01*}

6/2 • {*AI95-00419-01*} a composite type with a limited component;

6.1/2 • {*AI95-00419-01*} a derived type whose parent is limited and is not an interface.

6.a/2 **Ramification:** {*AI95-00419-01*} Limitedness is not inherited from interfaces; it must be explicitly specified when the parent is an interface.

6.b/2 **To be honest:** {*AI95-00419-01*} A derived type can become nonlimited if **limited** does not appear and the derivation takes place in the visible part of a child package, and the parent type is nonlimited as viewed from the private part or body of the child package.

6.c/2 **Reason:** {*AI95-00419-01*} We considered a rule where limitedness was always inherited from the parent for derived types, but in the case of a type whose parent is an interface, this meant that the first interface is treated differently than other interfaces. It also would have forced users to declare dummy nonlimited interfaces just to get the limitedness right. We also considered a syntax like **not limited** to specify nonlimitedness when the parent was limited, but that was unsavory. The rule given is more uniform and simpler to understand.

6.d/2 {*AI95-00419-01*} The rules for interfaces are asymmetrical, but the language is not: if the parent interface is limited, the presence of the word **limited** determines the limitedness, and nonlimited progenitors are illegal by the rules in 3.9.4. If the parent interface is nonlimited, the word **limited** is illegal by the rules in 3.4. The net effect is that the order of the interfaces doesn't matter.

7 {*nonlimited type*} Otherwise, the type is nonlimited.

8 [There are no predefined equality operators for a limited type.]

Implementation Requirements

8.1/2 {*AI95-00287-01*} {*AI95-00318-02*} For an aggregate of a limited type used to initialize an object as allowed above, the implementation shall not create a separate anonymous object for the aggregate. For a function_call of a type with a part that is of a task, protected, or explicitly limited record type that is used to initialize an object as allowed above, the implementation shall not create a separate return object (see 6.5) for the function_call. The aggregate or function_call shall be constructed directly in the new object.

8.a/2 **Discussion:** {*AI95-00318-02*} For a function_call, we only require *build-in-place*{*build-in-place* [partial]} for a limited type that would have been a return-by-reference type in Ada 95. We do this because we want to minimize disruption to Ada 95 implementations and users.

NOTES

9/2 14 {*AI95-00287-01*} {*AI95-00318-02*} While it is allowed to write initializations of limited objects, such initializations never copy a limited object. The source of such an assignment operation must be an aggregate or function_call, and such aggregates and function_calls must be built directly in the target object.

9.a/2 **To be honest:** This isn't quite true if the type can become nonlimited (see below); function_calls only are required to be build-in-place for “really” limited types.

Paragraphs 10 through 15 were deleted.

16 15 {*become nonlimited*} {*nonlimited type (becoming nonlimited)*} {*limited type (becoming nonlimited)*} As illustrated in 7.3.1, an untagged limited type can become nonlimited under certain circumstances.

16.a **Ramification:** Limited private types do not become nonlimited; instead, their full view can be nonlimited, which has a similar effect.

16.b It is important to remember that a single nonprivate type can be both limited and nonlimited in different parts of its scope. In other words, “limited” is a property that depends on where you are in the scope of the type. We don't call this a “view property” because there is no particular declaration to declare the nonlimited view.

16.c Tagged types never become nonlimited.

Examples

17 *Example of a package with a limited type:*

18
```
package IO_Package is
   type File_Name is limited private;
```

19
```
   procedure Open (F : in out File_Name);
   procedure Close(F : in out File_Name);
   procedure Read (F : in File_Name; Item : out Integer);
   procedure Write(F : in File_Name; Item : in  Integer);
private
   type File_Name is
      limited record
         Internal_Name : Integer := 0;
      end record;
end IO_Package;
```

20
```
package body IO_Package is
   Limit : constant := 200;
   type File_Descriptor is record  ...  end record;
   Directory : array (1 .. Limit) of File_Descriptor;
   ...
   procedure Open (F : in out File_Name) is  ...  end;
   procedure Close(F : in out File_Name) is  ...  end;
   procedure Read (F : in File_Name; Item : out Integer) is ... end;
   procedure Write(F : in File_Name; Item : in  Integer) is ... end;
begin
   ...
end IO_Package;
```

NOTES

16 *Notes on the example:* In the example above, an outside subprogram making use of IO_Package may obtain a file name by calling Open and later use it in calls to Read and Write. Thus, outside the package, a file name obtained from Open acts as a kind of password; its internal properties (such as containing a numeric value) are not known and no other operations (such as addition or comparison of internal names) can be performed on a file name. Most importantly, clients of the package cannot make copies of objects of type File_Name. 21

This example is characteristic of any case where complete control over the operations of a type is desired. Such packages serve a dual purpose. They prevent a user from making use of the internal structure of the type. They also implement the notion of an encapsulated data type where the only operations on the type are those given in the package specification. 22

{*AI95-00318-02*} The fact that the full view of File_Name is explicitly declared **limited** means that parameter passing will always be by reference and function results will always be built directly in the result object (see 6.2 and 6.5). 23/2

Extensions to Ada 83

{*extensions to Ada 83*} The restrictions in RM83-7.4.4(4), which disallowed **out** parameters of limited types in certain cases, are removed. 23.a

Wording Changes from Ada 83

Since limitedness and privateness are orthogonal in Ada 95 (and to some extent in Ada 83), this is now its own clause rather than being a subclause of 7.3, “Private Types and Private Extensions”. 23.b

Extensions to Ada 95

{*AI95-00287-01*} {*AI95-00318-02*} {*extensions to Ada 95*} Limited types now have an assignment operation, but its use is restricted such that all uses are build-in-place. This is accomplished by restricting uses to aggregates and function_calls. Aggregates were not allowed to have a limited type in Ada 95, which causes a compatibility issue discussed in 4.3, “Aggregates”. Compatibility issues with return statements for limited function_calls are discussed in 6.5, “Return Statements”. 23.c/2

Wording Changes from Ada 95

{*AI95-00411-01*} {*AI95-00419-01*} Rewrote the definition of limited to ensure that interfaces are covered, but that limitedness is not inherited from interfaces. Derived types that explicitly include **limited** are now also covered. 23.d/2

7.6 User-Defined Assignment and Finalization

[{*user-defined assignment*} {*assignment (user-defined)*} Three kinds of actions are fundamental to the manipulation of objects: initialization, finalization, and assignment. Every object is initialized, either explicitly or by default, after being created (for example, by an object_declaration or allocator). Every object is finalized before being destroyed (for example, by leaving a subprogram_body containing an 1 object_declaration, or by a call to an instance of Unchecked_Deallocation). An assignment operation is used as part of assignment_statements, explicit initialization, parameter passing, and other operations. {*constructor: See initialization*} {*constructor: See Initialize*} {*destructor: See finalization*}

2 Default definitions for these three fundamental operations are provided by the language, but {*controlled type*} a *controlled* type gives the user additional control over parts of these operations. {*Initialize*} {*Finalize*} {*Adjust*} In particular, the user can define, for a controlled type, an Initialize procedure which is invoked immediately after the normal default initialization of a controlled object, a Finalize procedure which is invoked immediately before finalization of any of the components of a controlled object, and an Adjust procedure which is invoked as the last step of an assignment to a (nonlimited) controlled object.]

2.a **Glossary entry:** {*Controlled type*} A controlled type supports user-defined assignment and finalization. Objects are always finalized before being destroyed.

2.b/2 **Ramification:** {*AI95-00114-01*} {*AI95-00287-01*} Here's the basic idea of initialization, value adjustment, and finalization, whether or not user defined: When an object is created, if it is explicitly assigned an initial value, the object is either built-in-place from an aggregate or function call (in which case neither Adjust nor Initialize is applied), or the assignment copies and adjusts the initial value. Otherwise, Initialize is applied to it (except in the case of an aggregate as a whole). An assignment_statement finalizes the target before copying in and adjusting the new value. Whenever an object goes away, it is finalized. Calls on Initialize and Adjust happen bottom-up; that is, components first, followed by the containing object. Calls on Finalize happen top-down; that is, first the containing object, and then its components. These ordering rules ensure that any components will be in a well-defined state when Initialize, Adjust, or Finalize is applied to the containing object.

Static Semantics

3 The following language-defined library package exists:

```
4/1  {8652/0020} {AI95-00126-01} package Ada.Finalization is
         pragma Preelaborate(Finalization);
         pragma Remote_Types(Finalization);

5/2  {AI95-00161-01}     type Controlled is abstract tagged private;
         pragma Preelaborable_Initialization(Controlled);

6/2  {AI95-00348-01}     procedure Initialize (Object : in out Controlled) is null;
         procedure Adjust     (Object : in out Controlled) is null;
         procedure Finalize   (Object : in out Controlled) is null;

7/2  {AI95-00161-01}     type Limited_Controlled is abstract tagged limited private;
         pragma Preelaborable_Initialization(Limited_Controlled);

8/2  {AI95-00348-01}     procedure Initialize (Object : in out Limited_Controlled) is
     null;
         procedure Finalize   (Object : in out Limited_Controlled) is null;
     private
         ... -- not specified by the language
     end Ada.Finalization;
```

9/2 {*AI95-00348-01*} {*controlled type*} A controlled type is a descendant of Controlled or Limited_Controlled. The predefined "=" operator of type Controlled always returns True, [since this operator is incorporated into the implementation of the predefined equality operator of types derived from Controlled, as explained in 4.5.2.] The type Limited_Controlled is like Controlled, except that it is limited and it lacks the primitive subprogram Adjust.

9.a **Discussion:** We say "nonlimited controlled type" (rather than just "controlled type"; when we want to talk about descendants of Controlled only.

9.b **Reason:** We considered making Adjust and Finalize abstract. However, a reasonable coding convention is e.g. for Finalize to always call the parent's Finalize after doing whatever work is needed for the extension part. (Unlike CLOS, we have no way to do that automatically in Ada 95.) For this to work, Finalize cannot be abstract. In a generic unit, for a generic formal abstract derived type whose ancestor is Controlled or Limited_Controlled, calling the ancestor's Finalize would be illegal if it were abstract, even though the actual type might have a concrete version.

9.c Types Controlled and Limited_Controlled are abstract, even though they have no abstract primitive subprograms. It is not clear that they need to be abstract, but there seems to be no harm in it, and it might make an implementation's life easier to know that there are no objects of these types — in case the implementation wishes to make them “magic” in some way.

9.d/2 {*AI95-00251-01*} For Ada 2005, we considered making these types interfaces. That would have the advantage of allowing them to be added to existing trees. But that was rejected both because it would cause massive disruption to existing implementations, and because it would be very incompatible due to the "no hidden interfaces" rule. The latter rule would prevent a tagged private type from being completed with a derivation from Controlled or Limited_Controlled — a very common idiom.

9.1/2 {*AI95-00360-01*} A type is said to *need finalization* if:{*needs finalization*} {*type (needs finalization)*}

- 9.2/2 it is a controlled type, a task type or a protected type; or
- 9.3/2 it has a component that needs finalization; or
- 9.4/2 it is a limited type that has an access discriminant whose designated type needs finalization; or
- 9.5/2 it is one of a number of language-defined types that are explicitly defined to need finalization.

9.e/2 **Ramification:** The fact that a type needs finalization does not require it to be implemented with a controlled type. It just has to be recognized by the No_Nested_Finalization restriction.

9.f/2 This property is defined for the type, not for a particular view. That's necessary as restrictions look in private parts to enforce their restrictions; the point is to eliminate all controlled parts, not just ones that are visible.

Dynamic Semantics

10/2 {*AI95-00373-01*} During the elaboration or evaluation of a construct that causes an object to be initialized by default, for every controlled subcomponent of the object that is not assigned an initial value (as defined in 3.3.1), Initialize is called on that subcomponent. Similarly, if the object that is initialized by default as a whole is controlled, Initialize is called on the object.

11/2 {*8652/0021*} {*AI95-00182-01*} {*AI95-00373-01*} For an extension_aggregate whose ancestor_part is a subtype_mark denoting a controlled subtype, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract.

11.a **Discussion:** Example:

11.b
```
type T1 is new Controlled with
    record
        ... -- some components might have defaults
    end record;
```

11.c
```
type T2 is new Controlled with
    record
        X : T1; -- no default
        Y : T1 := ...; -- default
    end record;
```

11.d
```
A : T2;
B : T2 := ...;
```

11.e As part of the elaboration of A's declaration, A.Y is assigned a value; therefore Initialize is not applied to A.Y. Instead, Adjust is applied to A.Y as part of the assignment operation. Initialize is applied to A.X and to A, since those objects are not assigned an initial value. The assignment to A.Y is not considered an assignment to A.

11.f For the elaboration of B's declaration, Initialize is not called at all. Instead the assignment adjusts B's value; that is, it applies Adjust to B.X, B.Y, and B.

11.f.1/2 {*8652/0021*} {*AI95-00182-01*} {*AI95-00373-01*} The ancestor_part of an extension_aggregate, <> in aggregates, and the return object of an extended_return_statement are handled similarly.

12 Initialize and other initialization operations are done in an arbitrary order, except as follows. Initialize is applied to an object after initialization of its subcomponents, if any [(including both implicit initialization and Initialize calls)]. If an object has a component with an access discriminant constrained by a per-object expression, Initialize is applied to this component after any components that do not have such discriminants. For an object with several components with such a discriminant, Initialize is applied to them in order of their component_declarations. For an allocator, any task activations follow all calls on Initialize.

12.a **Reason:** The fact that Initialize is done for subcomponents first allows Initialize for a composite object to refer to its subcomponents knowing they have been properly initialized.

12.b The fact that Initialize is done for components with access discriminants after other components allows the Initialize operation for a component with a self-referential access discriminant to assume that other components of the enclosing object have already been properly initialized. For multiple such components, it allows some predictability.

13 {*assignment operation*} When a target object with any controlled parts is assigned a value, [either when created or in a subsequent assignment_statement,] the *assignment operation* proceeds as follows:

14 • The value of the target becomes the assigned value.

15 • {*adjusting the value of an object*} {*adjustment*} The value of the target is *adjusted.*

15.a **Ramification:** If any parts of the object are controlled, abort is deferred during the assignment operation.

16 {*adjusting the value of an object*} {*adjustment*} To adjust the value of a [(nonlimited)] composite object, the values of the components of the object are first adjusted in an arbitrary order, and then, if the object is controlled, Adjust is called. Adjusting the value of an elementary object has no effect[, nor does adjusting the value of a composite object with no controlled parts.]

16.a **Ramification:** Adjustment is never performed for values of a by-reference limited type, since these types do not support copying.

16.b **Reason:** The verbiage in the Initialize rule about access discriminants constrained by per-object expressions is not necessary here, since such types are limited, and therefore are never adjusted.

17 {*execution (assignment_statement)* [partial]} For an assignment_statement, [after the name and expression have been evaluated, and any conversion (including constraint checking) has been done,] an anonymous object is created, and the value is assigned into it; [that is, the assignment operation is applied]. [(Assignment includes value adjustment.)] The target of the assignment_statement is then finalized. The value of the anonymous object is then assigned into the target of the assignment_statement. Finally, the anonymous object is finalized. [As explained below, the implementation may eliminate the intermediate anonymous object, so this description subsumes the one given in 5.2, “Assignment Statements”.]

17.a **Reason:** An alternative design for user-defined assignment might involve an Assign operation instead of Adjust:

17.b
```
procedure Assign(Target : in out Controlled; Source : in out Controlled);
```

17.c Or perhaps even a syntax like this:

17.d
```
procedure ":="(Target : in out Controlled; Source : in out Controlled);
```

17.e Assign (or ":=") would have the responsibility of doing the copy, as well as whatever else is necessary. This would have the advantage that the Assign operation knows about both the target and the source at the same time — it would be possible to do things like reuse storage belonging to the target, for example, which Adjust cannot do. However, this sort of design would not work in the case of unconstrained discriminated variables, because there is no way to change the discriminants individually. For example:

17.f
```
type Mutable(D : Integer := 0) is
    record
        X : Array_Of_Controlled_Things(1..D);
        case D is
            when 17 => Y : Controlled_Thing;
            when others => null;
        end D;
    end record;
```

17.g An assignment to an unconstrained variable of type Mutable can cause some of the components of X, and the component Y, to appear and/or disappear. There is no way to write the Assign operation to handle this sort of case.

17.h Forbidding such cases is not an option — it would cause generic contract model violations.

Implementation Requirements

17.1/2 {*8652/0022*} {*AI95-00083-01*} {*AI95-00318-02*} For an aggregate of a controlled type whose value is assigned, other than by an assignment_statement, the implementation shall not create a separate anonymous object for the aggregate. The aggregate value shall be constructed directly in the target of the assignment operation and Adjust is not called on the target object.

Reason: {*AI95-00318-02*} {*build-in-place* [partial]} This *build-in-place* requirement is necessary to prevent elaboration problems with deferred constants of controlled types. Consider: 17.h.1/2

17.h.2/1
```
package P is
   type Dyn_String is private;
   Null_String : constant Dyn_String;
   ...
private
   type Dyn_String is new Ada.Finalization.Controlled with ...
   procedure Finalize(X : in out Dyn_String);
   procedure Adjust(X : in out Dyn_String);

   Null_String : constant Dyn_String :=
      (Ada.Finalization.Controlled with ...);
   ...
end P;
```

When Null_String is elaborated, the bodies of Finalize and Adjust clearly have not been elaborated. Without this rule, this declaration would necessarily raise Program_Error (unless the permissions given below are used by the implementation). 17.h.3/1

Ramification: An aggregate used in the return expression of a simple_return_statement has to be built-in-place in the anonymous return object, as this is similar to an object declaration. (This is a change from Ada 95, but it is not an inconsistency as it only serves to restrict implementation choices.) But this only covers the aggregate; a separate anonymous return object can still be used unless it too is required to be built-in-place (see 7.5). 17.i/2

Implementation Permissions

An implementation is allowed to relax the above rules [(for nonlimited controlled types)] in the following ways: 18

Proof: The phrase “for nonlimited controlled types” follows from the fact that all of the following permissions apply to cases involving assignment. It is important because the programmer can count on a stricter semantics for limited controlled types. 18.a

- For an assignment_statement that assigns to an object the value of that same object, the implementation need not do anything. 19

Ramification: In other words, even if an object is controlled and a combination of Finalize and Adjust on the object might have a net side effect, they need not be performed. 19.a

- For an assignment_statement for a noncontrolled type, the implementation may finalize and assign each component of the variable separately (rather than finalizing the entire variable and assigning the entire new value) unless a discriminant of the variable is changed by the assignment. 20

Reason: For example, in a slice assignment, an anonymous object is not necessary if the slice is copied component-by-component in the right direction, since array types are not controlled (although their components may be). Note that the direction, and even the fact that it's a slice assignment, can in general be determined only at run time. 20.a

- {*AI95-00147-01*} For an aggregate or function call whose value is assigned into a target object, the implementation need not create a separate anonymous object if it can safely create the value of the aggregate or function call directly in the target object. Similarly, for an assignment_-statement, the implementation need not create an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose storage cannot overlap with the target. If the source object might overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object). 21/2

Ramification: In the aggregate case, only one value adjustment is necessary, and there is no anonymous object to be finalized. 21.a

{*AI95-00147-01*} Similarly, in the function call case, the anonymous object can be eliminated. Note, however, that Adjust must be called directly on the target object as the last step of the assignment, since some of the subcomponents may be self-referential or otherwise position-dependent. This Adjust can be eliminated only by using one of the following permissions. 21.b/2

22/2 {*AI95-00147-01*} Furthermore, an implementation is permitted to omit implicit Initialize, Adjust, and Finalize calls and associated assignment operations on an object of a nonlimited controlled type provided that:

23/2 • any omitted Initialize call is not a call on a user-defined Initialize procedure, and

23.a/2 **To be honest:** This does not apply to any calls to a user-defined Initialize routine that happen to occur in an Adjust or Finalize routine. It is intended that it is never necessary to look inside of an Adjust or Finalize routine to determine if the call can be omitted.

23.b/2 **Reason:** We don't want to eliminate objects for which the Initialize might have side effects (such as locking a resource).

24/2 • any usage of the value of the object after the implicit Initialize or Adjust call and before any subsequent Finalize call on the object does not change the external effect of the program, and

25/2 • after the omission of such calls and operations, any execution of the program that executes an Initialize or Adjust call on an object or initializes an object by an aggregate will also later execute a Finalize call on the object and will always do so prior to assigning a new value to the object, and

26/2 • the assignment operations associated with omitted Adjust calls are also omitted.

27/2 This permission applies to Adjust and Finalize calls even if the implicit calls have additional external effects.

27.a/2 **Reason:** The goal of the above permissions is to allow typical dead assignment and dead variable removal algorithms to work for nonlimited controlled types. We require that “pairs” of Initialize/Adjust/Finalize operations are removed. (These aren't always pairs, which is why we talk about “any execution of the program”.)

Extensions to Ada 83

27.b {*extensions to Ada 83*} Controlled types and user-defined finalization are new to Ada 95. (Ada 83 had finalization semantics only for masters of tasks.)

Extensions to Ada 95

27.c/2 {*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Types Controlled and Limited_Controlled now have Preelaborable_Initialization, so that objects of types derived from these types can be used in preelaborated packages.

Wording Changes from Ada 95

27.d/2 {*8652/0020*} {*AI95-00126-01*} **Corrigendum:** Clarified that Ada.Finalization is a remote types package.

27.e/2 {*8652/0021*} {*AI95-00182-01*} **Corrigendum:** Added wording to clarify that the default initialization (whatever it is) of an ancestor part is used.

27.f/2 {*8652/0022*} {*AI95-00083-01*} **Corrigendum:** Clarified that Adjust is never called on an aggregate used for the initialization of an object or subaggregate, or passed as a parameter.

27.g/2 {*AI95-00147-01*} Additional optimizations are allowed for nonlimited controlled types. These allow traditional dead variable elimination to be applied to such types.

27.h/2 {*AI95-00318-02*} Corrected the build-in-place requirement for controlled aggregates to be consistent with the requirements for limited types.

27.i/2 {*AI95-00348-01*} The operations of types Controlled and Limited_Controlled are now declared as null procedures (see 6.7) to make the semantics clear (and to provide a good example of what null procedures can be used for).

27.j/2 {*AI95-00360-01*} Types that need finalization are defined; this is used by the No_Nested_Finalization restriction (see D.7, “Tasking Restrictions”).

27.k/2 {*AI95-00373-01*} Generalized the description of objects that have Initialize called for them to say that it is done for all objects that are initialized by default. This is needed so that all of the new cases are covered.

7.6.1 Completion and Finalization

[This subclause defines *completion* and *leaving* of the execution of constructs and entities. A *master* is the execution of a construct that includes finalization of local objects after it is complete (and after waiting for any local tasks — see 9.3), but before leaving. Other constructs and entities are left immediately upon completion. {*cleanup: See finalization*} {*destructor: See finalization*}] 1

Dynamic Semantics

{*AI95-00318-02*} {*completion and leaving (completed and left)*} {*completion (run-time concept)*} The execution of a construct or entity is *complete* when the end of that execution has been reached, or when a transfer of control (see 5.1) causes it to be abandoned. {*normal completion*} {*completion (normal)*} {*abnormal completion*} {*completion (abnormal)*} Completion due to reaching the end of execution, or due to the transfer of control of an exit_statement, return statement, goto_statement, or requeue_statement or of the selection of a terminate_alternative is *normal completion*. Completion is *abnormal* otherwise [— when control is transferred out of a construct due to abort or the raising of an exception]. 2/2

Discussion: Don't confuse the run-time concept of completion with the compile-time concept of completion defined in 3.11.1. 2.a

{*AI95-00162-01*} {*AI95-00416-01*} {*leaving*} {*left*} After execution of a construct or entity is complete, it is *left*, meaning that execution continues with the next action, as defined for the execution that is taking place. {*master*} Leaving an execution happens immediately after its completion, except in the case of a *master*: the execution of a body other than a package_body; the execution of a statement; or the evaluation of an expression, function_call, or range that is not part of an enclosing expression, function_call, range, or simple_statement other than a simple_return_statement. A master is finalized after it is complete, and before it is left. 3/2

Reason: {*AI95-00162-01*} {*AI95-00416-01*} Expressions and statements are masters so that objects created by subprogram calls (in aggregates, allocators for anonymous access-to-object types, and so on) are finalized and have their tasks awaited before the expressions or statements are left. Note that expressions like the condition of an if_statement are masters, because they are not enclosed by a simple_statement. Similarly, a function_call which is renamed is a master, as it is not in a simple_statement. 3.a/2

{*AI95-00416-01*} We have to include function_calls in the contexts that do not cause masters to occur so that expressions contained in a function_call (that is not part of an expression or simple_statement) do not individually become masters. We certainly do not want the parameter expressions of a function_call to be separate masters, as they would then be finalized before the function is called. 3.b/2

Ramification: {*AI95-00416-01*} The fact that a function_call is a master does not change the accessibility of the return object denoted by the function_call; that depends on the use of the function_call. The function_call is the master of any short-lived entities (such as aggregates used as parameters of types with task or controlled parts). 3.c/2

{*finalization (of a master)*} For the *finalization* of a master, dependent tasks are first awaited, as explained in 9.3. Then each object whose accessibility level is the same as that of the master is finalized if the object was successfully initialized and still exists. [These actions are performed whether the master is left by reaching the last statement or via a transfer of control.] When a transfer of control causes completion of an execution, each included master is finalized in order, from innermost outward. 4

Ramification: As explained in 3.10.2, the set of objects with the same accessibility level as that of the master includes objects declared immediately within the master, objects declared in nested packages, objects created by allocators (if the ultimate ancestor access type is declared in one of those places) and subcomponents of all of these things. If an object was already finalized by Unchecked_Deallocation, then it is not finalized again when the master is left. 4.a

Note that any object whose accessibility level is deeper than that of the master would no longer exist; those objects would have been finalized by some inner master. Thus, after leaving a master, the only objects yet to be finalized are those whose accessibility level is less deep than that of the master. 4.b

To be honest: Subcomponents of objects due to be finalized are not finalized by the finalization of the master; they are finalized by the finalization of the containing object. 4.c

4.d **Reason:** We need to finalize subcomponents of objects even if the containing object is not going to get finalized because it was not fully initialized. But if the containing object is finalized, we don't want to require repeated finalization of the subcomponents, as might normally be implied by the recursion in finalization of a master and the recursion in finalization of an object.

4.e **To be honest:** Formally, completion and leaving refer to executions of constructs or entities. However, the standard sometimes (informally) refers to the constructs or entities whose executions are being completed. Thus, for example, "the subprogram call or task is complete" really means "*the execution of* the subprogram call or task is complete."

5 {*finalization (of an object)* [distributed]} For the *finalization* of an object:

6 • If the object is of an elementary type, finalization has no effect;

7 • If the object is of a controlled type, the Finalize procedure is called;

8 • If the object is of a protected type, the actions defined in 9.4 are performed;

9/2 • {*AI95-00416-01*} If the object is of a composite type, then after performing the above actions, if any, every component of the object is finalized in an arbitrary order, except as follows: if the object has a component with an access discriminant constrained by a per-object expression, this component is finalized before any components that do not have such discriminants; for an object with several components with such a discriminant, they are finalized in the reverse of the order of their component_declarations;

9.a **Reason:** This allows the finalization of a component with an access discriminant to refer to other components of the enclosing object prior to their being finalized.

9.1/2 • {*AI95-00416-01*} If the object has coextensions (see 3.10.2), each coextension is finalized after the object whose access discriminant designates it.

10 {*execution (instance of Unchecked_Deallocation)* [partial]} Immediately before an instance of Unchecked_Deallocation reclaims the storage of an object, the object is finalized. [If an instance of Unchecked_Deallocation is never applied to an object created by an allocator, the object will still exist when the corresponding master completes, and it will be finalized then.]

11/2 {*AI95-00280-01*} The order in which the finalization of a master performs finalization of objects is as follows: Objects created by declarations in the master are finalized in the reverse order of their creation. For objects that were created by allocators for an access type whose ultimate ancestor is declared in the master, this rule is applied as though each such object that still exists had been created in an arbitrary order at the first freezing point (see 13.14) of the ultimate ancestor type; the finalization of these objects is called the *finalization of the collection*{*finalization of the collection*} {*collection (finalization of)*} . After the finalization of a master is complete, the objects finalized as part of its finalization cease to *exist*, as do any types and subtypes defined and created within the master.{*exist (cease to)* [partial]} {*cease to exist (object)* [partial]} {*cease to exist (type)*}

11.a **Reason:** Note that we talk about the type of the allocator here. There may be access values of a (general) access type pointing at objects created by allocators for some other type; these are not finalized at this point.

11.b The freezing point of the ultimate ancestor access type is chosen because before that point, pool elements cannot be created, and after that point, access values designating (parts of) the pool elements can be created. This is also the point after which the pool object cannot have been declared. We don't want to finalize the pool elements until after anything finalizing objects that contain access values designating them. Nor do we want to finalize pool elements after finalizing the pool object itself.

11.c **Ramification:** Finalization of allocated objects is done according to the (ultimate ancestor) allocator type, not according to the storage pool in which they are allocated. Pool finalization might reclaim storage (see 13.11, "Storage Management"), but has nothing (directly) to do with finalization of the pool elements.

11.d Note that finalization is done only for objects that still exist; if an instance of Unchecked_Deallocation has already gotten rid of a given pool element, that pool element will not be finalized when the master is left.

11.e Note that a deferred constant declaration does not create the constant; the full constant declaration creates it. Therefore, the order of finalization depends on where the full constant declaration occurs, not the deferred constant declaration.

11.f An imported object is not created by its declaration. It is neither initialized nor finalized.

Implementation Note: An implementation has to ensure that the storage for an object is not reclaimed when references to the object are still possible (unless, of course, the user explicitly requests reclamation via an instance of Unchecked_Deallocation). This implies, in general, that objects cannot be deallocated one by one as they are finalized; a subsequent finalization might reference an object that has been finalized, and that object had better be in its (well-defined) finalized state. 11.g

{*AI95-00256-01*} {*execution (assignment_statement)* [partial]} The target of an assignment_statement is finalized before copying in the new value, as explained in 7.6. 12/2

{*8652/0021*} {*AI95-00182-01*} {*AI95-00162-01*} The master of an object is the master enclosing its creation whose accessibility level (see 3.10.2) is equal to that of the object. 13/2

This paragraph was deleted.{*AI95-00162-01*} 13.a/2

This paragraph was deleted. 13.b/2

This paragraph was deleted. 13.c/2

Reason: {*AI95-00162-01*} This effectively imports all of the special rules for the accessibility level of renames, allocators, and so on, and applies them to determine where objects created in them are finalized. For instance, the master of a rename of a subprogram is that of the renamed subprogram. 13.d/2

{*8652/0023*} {*AI95-00169-01*} {*AI95-00162-01*} In the case of an expression that is a master, finalization of any (anonymous) objects occurs as the final part of evaluation of the expression. 13.1/2

Bounded (Run-Time) Errors

{*8652/0023*} {*AI95-00169-01*} {*bounded error (cause)* [partial]} It is a bounded error for a call on Finalize or Adjust that occurs as part of object finalization or assignment to propagate an exception. The possible consequences depend on what action invoked the Finalize or Adjust operation: 14/1

Ramification: It is not a bounded error for Initialize to propagate an exception. If Initialize propagates an exception, then no further calls on Initialize are performed, and those components that have already been initialized (either explicitly or by default) are finalized in the usual way. 14.a

{*8652/0023*} {*AI95-00169-01*} It also is not a bounded error for an explicit call to Finalize or Adjust to propagate an exception. We do not want implementations to have to treat explicit calls to these routines specially. 14.a.1/1

- {*Program_Error (raised by failure of run-time check)*} For a Finalize invoked as part of an assignment_statement, Program_Error is raised at that point. 15
- {*8652/0024*} {*AI95-00193-01*} {*AI95-00256-01*} For an Adjust invoked as part of assignment operations other than those invoked as part of an assignment_statement, other adjustments due to be performed might or might not be performed, and then Program_Error is raised. During its propagation, finalization might or might not be applied to objects whose Adjust failed. {*Program_Error (raised by failure of run-time check)*} For an Adjust invoked as part of an assignment_statement, any other adjustments due to be performed are performed, and then Program_Error is raised. 16/2

Reason: {*8652/0024*} {*AI95-00193-01*} {*AI95-00256-01*} In the case of assignments that are part of initialization, there is no need to complete all adjustments if one propagates an exception, as the object will immediately be finalized. So long as a subcomponent is not going to be finalized, it need not be adjusted, even if it is initialized as part of an enclosing composite assignment operation for which some adjustments are performed. However, there is no harm in an implementation making additional Adjust calls (as long as any additional components that are adjusted are also finalized), so we allow the implementation flexibility here. On the other hand, for an assignment_statement, it is important that all adjustments be performed, even if one fails, because all controlled subcomponents are going to be finalized. Other kinds of assignment are more like initialization than assignment_statements, so we include them as well in the permission. 16.a/2

Ramification: {*8652/0024*} {*AI95-00193-01*} Even if an Adjust invoked as part of the initialization of a controlled object propagates an exception, objects whose initialization (including any Adjust or Initialize calls) successfully completed will be finalized. The permission above only applies to objects whose Adjust failed. Objects for which Adjust was never even invoked must not be finalized. 16.a.1/1

- {*Program_Error (raised by failure of run-time check)*} For a Finalize invoked as part of a call on an instance of Unchecked_Deallocation, any other finalizations due to be performed are performed, and then Program_Error is raised. 17

17.a.1/1 **Discussion:** {*8652/0104*} {*AI95-00179-01*} The standard does not specify if storage is recovered in this case. If storage is not recovered (and the object continues to exist), Finalize may be called on the object again (when the allocator's master is finalized).

17.1/1
- {*8652/0023*} {*AI95-00169-01*} {*Program_Error (raised by failure of run-time check)*} For a Finalize invoked as part of the finalization of the anonymous object created by a function call or aggregate, any other finalizations due to be performed are performed, and then Program_Error is raised.

17.2/1
- {*8652/0023*} {*AI95-00169-01*} {*Program_Error (raised by failure of run-time check)*} For a Finalize invoked due to reaching the end of the execution of a master, any other finalizations associated with the master are performed, and Program_Error is raised immediately after leaving the master.

18/2
- {*AI95-00318-02*} {*Program_Error (raised by failure of run-time check)*} For a Finalize invoked by the transfer of control of an exit_statement, return statement, goto_statement, or requeue_statement, Program_Error is raised no earlier than after the finalization of the master being finalized when the exception occurred, and no later than the point where normal execution would have continued. Any other finalizations due to be performed up to that point are performed before raising Program_Error.

18.a **Ramification:** For example, upon leaving a block_statement due to a goto_statement, the Program_Error would be raised at the point of the target statement denoted by the label, or else in some more dynamically nested place, but not so nested as to allow an exception_handler that has visibility upon the finalized object to handle it. For example,

18.b
```
procedure Main is
begin
    <<The_Label>>
    Outer_Block_Statement : declare
        X : Some_Controlled_Type;
    begin
        Inner_Block_Statement : declare
            Y : Some_Controlled_Type;
            Z : Some_Controlled_Type;
        begin
            goto The_Label;
        exception
            when Program_Error => ... -- Handler number 1.
        end;
    exception
        when Program_Error => ... -- Handler number 2.
    end;
exception
    when Program_Error => ... -- Handler number 3.
end Main;
```

18.c The goto_statement will first cause Finalize(Y) to be called. Suppose that Finalize(Y) propagates an exception. Program_Error will be raised after leaving Inner_Block_Statement, but before leaving Main. Thus, handler number 1 cannot handle this Program_Error; it will be handled either by handler number 2 or handler number 3. If it is handled by handler number 2, then Finalize(Z) will be done before executing the handler. If it is handled by handler number 3, then Finalize(Z) and Finalize(X) will both be done before executing the handler.

19
- For a Finalize invoked by a transfer of control that is due to raising an exception, any other finalizations due to be performed for the same master are performed; Program_Error is raised immediately after leaving the master.

19.a **Ramification:** If, in the above example, the goto_statement were replaced by a raise_statement, then the Program_Error would be handled by handler number 2, and Finalize(Z) would be done before executing the handler.

19.b **Reason:** We considered treating this case in the same way as the others, but that would render certain exception_handlers useless. For example, suppose the only exception_handler is one for **others** in the main subprogram. If some deeply nested call raises an exception, causing some Finalize operation to be called, which then raises an exception, then normal execution “would have continued” at the beginning of the exception_handler. Raising Program_Error at that point would cause that handler's code to be skipped. One would need two nested exception_handlers to be sure of catching such cases!

19.c On the other hand, the exception_handler for a given master should not be allowed to handle exceptions raised during finalization of that master.

- For a Finalize invoked by a transfer of control due to an abort or selection of a terminate alternative, the exception is ignored; any other finalizations due to be performed are performed. 20

Ramification: This case includes an asynchronous transfer of control. 20.a

To be honest: {*Program_Error (raised by failure of run-time check)*} This violates the general principle that it is always possible for a bounded error to raise Program_Error (see 1.1.5, “Classification of Errors”). 20.b

NOTES

17 The rules of Section 10 imply that immediately prior to partition termination, Finalize operations are applied to library-level controlled objects (including those created by allocators of library-level access types, except those already finalized). This occurs after waiting for library-level tasks to terminate. 21

Discussion: We considered defining a pragma that would apply to a controlled type that would suppress Finalize operations for library-level objects of the type upon partition termination. This would be useful for types whose finalization actions consist of simply reclaiming global heap storage, when this is already provided automatically by the environment upon program termination. 21.a

18 A constant is only constant between its initialization and finalization. Both initialization and finalization are allowed to change the value of a constant. 22

19 Abort is deferred during certain operations related to controlled types, as explained in 9.8. Those rules prevent an abort from causing a controlled object to be left in an ill-defined state. 23

20 The Finalize procedure is called upon finalization of a controlled object, even if Finalize was called earlier, either explicitly or as part of an assignment; hence, if a controlled type is visibly controlled (implying that its Finalize primitive is directly callable), or is nonlimited (implying that assignment is allowed), its Finalize procedure should be designed to have no ill effect if it is applied a second time to the same object. 24

Discussion: Or equivalently, a Finalize procedure should be “idempotent”; applying it twice to the same object should be equivalent to applying it once. 24.a

Reason: A user-written Finalize procedure should be idempotent since it can be called explicitly by a client (at least if the type is "visibly" controlled). Also, Finalize is used implicitly as part of the assignment_statement if the type is nonlimited, and an abort is permitted to disrupt an assignment_statement between finalizing the left-hand side and assigning the new value to it (an abort is not permitted to disrupt an assignment operation between copying in the new value and adjusting it). 24.b

Discussion: {*AI95-00287-01*} Either Initialize or Adjust, but not both, is applied to (almost) every controlled object when it is created: Initialize is done when no initial value is assigned to the object, whereas Adjust is done as part of assigning the initial value. The one exception is the object initialized by an aggregate (both the anonymous object created for an aggregate, or an object initialized by an aggregate that is built-in-place); Initialize is not applied to the aggregate as a whole, nor is the value of the aggregate or object adjusted. 24.c/2

{*assignment operation (list of uses)*} All of the following use the assignment operation, and thus perform value adjustment: 24.d

- the assignment_statement (see 5.2); 24.e
- explicit initialization of a stand-alone object (see 3.3.1) or of a pool element (see 4.8); 24.f
- default initialization of a component of a stand-alone object or pool element (in this case, the value of each component is assigned, and therefore adjusted, but the value of the object as a whole is not adjusted); 24.g
- {*AI95-00318-02*} function return, when the result is not built-in-place (adjustment of the result happens before finalization of the function); 24.h/2
- predefined operators (although the only one that matters is concatenation; see 4.5.3); 24.i
- generic formal objects of mode **in** (see 12.4); these are defined in terms of constant declarations; and 24.j
- {*AI95-00287-01*} aggregates (see 4.3), when the result is not built-in-place (in this case, the value of each component, and the parent part, for an extension_aggregate, is assigned, and therefore adjusted, but the value of the aggregate as a whole is not adjusted; neither is Initialize called); 24.k/2

The following also use the assignment operation, but adjustment never does anything interesting in these cases: 24.l

- By-copy parameter passing uses the assignment operation (see 6.4.1), but controlled objects are always passed by reference, so the assignment operation never does anything interesting in this case. If we were to allow by-copy parameter passing for controlled objects, we would need to make sure that the actual is finalized before doing the copy back for [**in**] **out** parameters. The finalization of the parameter itself needs to happen after the copy back (if any), similar to the finalization of an anonymous function return object or aggregate object. 24.m
- **For** loops use the assignment operation (see 5.5), but since the type of the loop parameter is never controlled, nothing interesting happens there, either. 24.n

24.n.1/2
- {*AI95-00318-02*} Objects initialized by function results and aggregates that are built-in-place. In this case, the assignment operation is never executed, and no adjustment takes place. While built-in-place is always allowed, it is required for some types — see 7.5 and 7.6 — and that's important since limited types have no Adjust to call.

24.o/2 *This paragraph was deleted.*{*AI95-00287-01*}

24.p Finalization of the parts of a protected object are not done as protected actions. It is possible (in pathological cases) to create tasks during finalization that access these parts in parallel with the finalization itself. This is an erroneous use of shared variables.

24.q **Implementation Note:** One implementation technique for finalization is to chain the controlled objects together on a per-task list. When leaving a master, the list can be walked up to a marked place. The links needed to implement the list can be declared (privately) in types Controlled and Limited_Controlled, so they will be inherited by all controlled types.

24.r Another implementation technique, which we refer to as the “PC-map” approach essentially implies inserting exception handlers at various places, and finalizing objects based on where the exception was raised.

24.s {*PC-map approach to finalization*} {*program-counter-map approach to finalization*} The PC-map approach is for the compiler/linker to create a map of code addresses; when an exception is raised, or abort occurs, the map can be consulted to see where the task was executing, and what finalization needs to be performed. This approach was given in the Ada 83 Rationale as a possible implementation strategy for exception handling — the map is consulted to determine which exception handler applies.

24.t If the PC-map approach is used, the implementation must take care in the case of arrays. The generated code will generally contain a loop to initialize an array. If an exception is raised part way through the array, the components that have been initialized must be finalized, and the others must not be finalized.

24.u It is our intention that both of these implementation methods should be possible.

Wording Changes from Ada 83

24.v Finalization depends on the concepts of completion and leaving, and on the concept of a master. Therefore, we have moved the definitions of these concepts here, from where they used to be in Section 9. These concepts also needed to be generalized somewhat. Task waiting is closely related to user-defined finalization; the rules here refer to the task-waiting rules of Section 9.

Wording Changes from Ada 95

24.w/2 {*8652/0021*} {*AI95-00182-01*} **Corrigendum:** Fixed the wording to say that anonymous objects aren't finalized until the object can't be used anymore.

24.x/2 {*8652/0023*} {*AI95-00169-01*} **Corrigendum:** Added wording to clarify what happens when Adjust or Finalize raises an exception; some cases had been omitted.

24.y/2 {*8652/0024*} {*AI95-00193-01*} {*AI95-00256-01*} **Corrigendum:** Stated that if Adjust raises an exception during initialization, nothing further is required. This is corrected in Ada 2005 to include all kinds of assignment other than assignment_statements.

24.z/2 {*AI95-00162-01*} {*AI95-00416-01*} Revised the definition of master to include expressions and statements, in order to cleanly define what happens for tasks and controlled objects created as part of a subprogram call. Having done that, all of the special wording to cover those cases is eliminated (at least until the Ada comments start rolling in).

24.aa/2 {*AI95-00280-01*} We define *finalization of the collection* here, so as to be able to conveniently refer to it in other rules (especially in 4.8, “Allocators”).

24.bb/2 {*AI95-00416-01*} Clarified that a coextension is finalized at the same time as the outer object. (This was intended for Ada 95, but since the concept did not have a name, it was overlooked.)

Section 8: Visibility Rules

1 [The rules defining the scope of declarations and the rules defining which identifiers, character_literals, and operator_symbols are visible at (or from) various places in the text of the program are described in this section. The formulation of these rules uses the notion of a declarative region.

2 As explained in Section 3, a declaration declares a view of an entity and associates a defining name with that view. The view comprises an identification of the viewed entity, and possibly additional properties. A usage name denotes a declaration. It also denotes the view declared by that declaration, and denotes the entity of that view. Thus, two different usage names might denote two different views of the same entity; in this case they denote the same entity.]

2.a **To be honest:** In some cases, a usage name that denotes a declaration does not denote the view declared by that declaration, nor the entity of that view, but instead denotes a view of the current instance of the entity, and denotes the current instance of the entity. This sometimes happens when the usage name occurs inside the declarative region of the declaration.

Wording Changes from Ada 83

2.b We no longer define the term “basic operation;” thus we no longer have to worry about the visibility of them. Since they were essentially always visible in Ada 83, this change has no effect. The reason for this change is that the definition in Ada 83 was confusing, and not quite correct, and we found it difficult to fix. For example, one wonders why an if_statement was not a basic operation of type Boolean. For another example, one wonders what it meant for a basic operation to be “inherent in” something. Finally, this fixes the problem addressed by AI83-00027/07.

8.1 Declarative Region

Static Semantics

1 {*declarative region (of a construct)*} For each of the following constructs, there is a portion of the program text called its *declarative region*, [within which nested declarations can occur]:

- 2 any declaration, other than that of an enumeration type, that is not a completion [of a previous declaration];
- 3 a block_statement;
- 4 a loop_statement;
- 4.1/2 {*AI95-00318-02*} an extended_return_statement;
- 5 an accept_statement;
- 6 an exception_handler.

7 The declarative region includes the text of the construct together with additional text determined [(recursively)], as follows:

- 8 If a declaration is included, so is its completion, if any.
- 9 If the declaration of a library unit [(including Standard — see 10.1.1)] is included, so are the declarations of any child units [(and their completions, by the previous rule)]. The child declarations occur after the declaration.
- 10 If a body_stub is included, so is the corresponding subunit.
- 11 If a type_declaration is included, then so is a corresponding record_representation_clause, if any.

11.a **Reason:** This is so that the component_declarations can be directly visible in the record_representation_clause.

12 The declarative region of a declaration is also called the *declarative region* of any view or entity declared by the declaration.

12.a **Reason:** The constructs that have declarative regions are the constructs that can have declarations nested inside them. Nested declarations are declared in that declarative region. The one exception is for enumeration literals; although they are nested inside an enumeration type declaration, they behave as if they were declared at the same level as the type.

12.b **To be honest:** A declarative region does not include parent_unit_names.

12.c **Ramification:** A declarative region does not include context_clauses.

13 {*occur immediately within*} {*immediately within*} {*within (immediately)*} {*immediately enclosing*} {*enclosing (immediately)*} A declaration occurs *immediately within* a declarative region if this region is the innermost declarative region that encloses the declaration (the *immediately enclosing* declarative region), not counting the declarative region (if any) associated with the declaration itself.

13.a **Discussion:** Don't confuse the declarative region of a declaration with the declarative region in which it immediately occurs.

14 [{*local to*} A declaration is *local* to a declarative region if the declaration occurs immediately within the declarative region.] [An entity is *local* to a declarative region if the entity is declared by a declaration that is local to the declarative region.]

14.a **Ramification:** "Occurs immediately within" and "local to" are synonyms (when referring to declarations).

14.b Thus, “local to” applies to both declarations and entities, whereas “occurs immediately within” only applies to declarations. We use this term only informally; for cases where precision is required, we use the term "occurs immediately within", since it is less likely to cause confusion.

15 {*global to*} A declaration is *global* to a declarative region if the declaration occurs immediately within another declarative region that encloses the declarative region. An entity is *global* to a declarative region if the entity is declared by a declaration that is global to the declarative region.

NOTES

16 1 The children of a parent library unit are inside the parent's declarative region, even though they do not occur inside the parent's declaration or body. This implies that one can use (for example) "P.Q" to refer to a child of P whose defining name is Q, and that after "**use** P;" Q can refer (directly) to that child.

17 2 As explained above and in 10.1.1, “Compilation Units - Library Units”, all library units are descendants of Standard, and so are contained in the declarative region of Standard. They are *not* inside the declaration or body of Standard, but they *are* inside its declarative region.

18 3 For a declarative region that comes in multiple parts, the text of the declarative region does not contain any text that might appear between the parts. Thus, when a portion of a declarative region is said to extend from one place to another in the declarative region, the portion does not contain any text that might appear between the parts of the declarative region.

18.a **Discussion:** It is necessary for the things that have a declarative region to include anything that contains declarations (except for enumeration type declarations). This includes any declaration that has a profile (that is, subprogram_declaration, subprogram_body, entry_declaration, subprogram_renaming_declaration, formal_subprogram_declaration, access-to-subprogram type_declaration), anything that has a discriminant_part (that is, various kinds of type_declaration), anything that has a component_list (that is, record type_declaration and record extension type_declaration), and finally the declarations of task and protected units and packages.

Wording Changes from Ada 83

18.b It was necessary to extend Ada 83's definition of declarative region to take the following Ada 95 features into account:

18.c
- Child library units.

18.d
- Derived types/type extensions — we need a declarative region for inherited components and also for new components.

18.e
- All the kinds of types that allow discriminants.

18.f
- Protected units.

18.g
- Entries that have bodies instead of accept statements.

18.h
- The choice_parameter_specification of an exception_handler.

18.i
- The formal parameters of access-to-subprogram types.

18.j
- Renamings-as-body.

Discriminated and access-to-subprogram type declarations need a declarative region. Enumeration type declarations cannot have one, because you don't have to say "Color.Red" to refer to the literal Red of Color. For other type declarations, it doesn't really matter whether or not there is an associated declarative region, so for simplicity, we give one to all types except enumeration types. 18.k

We now say that an accept_statement has its own declarative region, rather than being part of the declarative region of the entry_declaration, so that declarative regions are properly nested regions of text, so that it makes sense to talk about "inner declarative regions," and "...extends to the end of a declarative region." Inside an accept_statement, the name of one of the parameters denotes the parameter_specification of the accept_statement, not that of the entry_declaration. If the accept_statement is nested within a block_statement, these parameter_specifications can hide declarations of the block_statement. The semantics of such cases was unclear in RM83. 18.l

To be honest: Unfortunately, we have the same problem for the entry name itself — it should denote the accept_statement, but accept_statements are not declarations. They should be, and they should hide the entry from all visibility within themselves. 18.m

Note that we can't generalize this to entry_bodies, or other bodies, because the declarative_part of a body is not supposed to contain (explicit) homographs of things in the declaration. It works for accept_statements only because an accept_statement does not have a declarative_part. 18.n

To avoid confusion, we use the term “local to” only informally in Ada 95. Even RM83 used the term incorrectly (see, for example, RM83-12.3(13)). 18.o

In Ada 83, (root) library units were inside Standard; it was not clear whether the declaration or body of Standard was meant. In Ada 95, they are children of Standard, and so occur immediately within Standard's declarative region, but not within either the declaration or the body. (See RM83-8.6(2) and RM83-10.1.1(5).) 18.p

Wording Changes from Ada 95

{*AI95-00318-02*} Extended_return_statement (see 6.5) is added to the list of constructs that have a declarative region. 18.q/2

8.2 Scope of Declarations

[For each declaration, the language rules define a certain portion of the program text called the *scope* of the declaration. The scope of a declaration is also called the scope of any view or entity declared by the declaration. Within the scope of an entity, and only there, there are places where it is legal to refer to the declared entity. These places are defined by the rules of visibility and overloading.] 1

Static Semantics

{*immediate scope (of a declaration)*} The *immediate scope* of a declaration is a portion of the declarative region immediately enclosing the declaration. The immediate scope starts at the beginning of the declaration, except in the case of an overloadable declaration, in which case the immediate scope starts just after the place where the profile of the callable entity is determined (which is at the end of the _specification for the callable entity, or at the end of the generic_instantiation if an instance). The immediate scope extends to the end of the declarative region, with the following exceptions: 2

Reason: The reason for making overloadable declarations with profiles special is to simplify compilation: until the compiler has determined the profile, it doesn't know which other declarations are homographs of this one, so it doesn't know which ones this one should hide. Without this rule, two passes over the _specification or generic_instantiation would be required to resolve names that denote things with the same name as this one. 2.a

- The immediate scope of a library_item includes only its semantic dependents. 3

Reason: Section 10 defines only a partial ordering of library_items. Therefore, it is a good idea to restrict the immediate scope (and the scope, defined below) to semantic dependents. 3.a

Consider also examples like this: 3.b

```
package P is end P;
```
3.c

```
package P.Q is
    I : Integer := 0;
end P.Q;
```
3.d

3.e/1
```
with P;
package R is
    package X renames P;
    J : Integer := X.Q.I; -- Illegal!
end R;
```

3.f The scope of P.Q does not contain R. Hence, neither P.Q nor X.Q are visible within R. However, the name R.X.Q would be visible in some other library unit where both R and P.Q are visible (assuming R were made legal by removing the offending declaration).

3.g/2 **Ramification:** {*AI95-00217-06*} This rule applies to limited views as well as "normal" library items. In that case, the semantic dependents are the units that have a limited_with_clause for the limited view.

4 • The immediate scope of a declaration in the private part of a library unit does not include the visible part of any public descendant of that library unit. {*descendant (relationship with scope)* [partial]}

4.a **Ramification:** In other words, a declaration in the private part can be visible within the visible part, private part and body of a private child unit. On the other hand, such a declaration can be visible within only the private part and body of a public child unit.

4.b **Reason:** The purpose of this rule is to prevent children from giving private information to clients.

4.c/2 **Ramification:** {*AI95-00231-01*} For a public child subprogram, this means that the parent's private part is not visible in the profile of the declaration and of the body. This is true even for subprogram_bodies that are not completions. For a public child generic unit, it means that the parent's private part is not visible in the generic_formal_part, as well as in the first list of basic_declarative_items (for a generic package), or the (syntactic) profile (for a generic subprogram).

5 {*visible part*} [The *visible part* of (a view of) an entity is a portion of the text of its declaration containing declarations that are visible from outside.] {*private part* [distributed]} The *private part* of (a view of) an entity that has a visible part contains all declarations within the declaration of (the view of) the entity, except those in the visible part; [these are not visible from outside. Visible and private parts are defined only for these kinds of entities: callable entities, other program units, and composite types.]

6 • {*visible part (of a view of a callable entity)* [partial]} The visible part of a view of a callable entity is its profile.

7 • {*visible part (of a view of a composite type)* [partial]} The visible part of a composite type other than a task or protected type consists of the declarations of all components declared [(explicitly or implicitly)] within the type_declaration.

8 • {*visible part (of a generic unit)* [partial]} The visible part of a generic unit includes the generic_formal_part. For a generic package, it also includes the first list of basic_declarative_items of the package_specification. For a generic subprogram, it also includes the profile.

8.a **Reason:** Although there is no way to reference anything but the formals from outside a generic unit, they are still in the visible part in the sense that the corresponding declarations in an instance can be referenced (at least in some cases). In other words, these declarations have an effect on the outside world. The visible part of a generic unit needs to be defined this way in order to properly support the rule that makes a parent's private part invisible within a public child's visible part.

8.b **Ramification:** The visible part of an instance of a generic unit is as defined for packages and subprograms; it is not defined in terms of the visible part of a generic unit.

9 • [The visible part of a package, task unit, or protected unit consists of declarations in the program unit's declaration other than those following the reserved word **private**, if any; see 7.1 and 12.7 for packages, 9.1 for task units, and 9.4 for protected units.]

10 {*scope (of a declaration)*} The scope of a declaration always contains the immediate scope of the declaration. In addition, for a given declaration that occurs immediately within the visible part of an outer declaration, or is a public child of an outer declaration, the scope of the given declaration extends to the end of the scope of the outer declaration, except that the scope of a library_item includes only its semantic dependents.

Ramification: Note the recursion. If a declaration appears in the visible part of a library unit, its scope extends to the end of the scope of the library unit, but since that only includes dependents of the declaration of the library unit, the scope of the inner declaration also only includes those dependents. If X renames library package P, which has a child Q, a with_clause mentioning P.Q is necessary to be able to refer to X.Q, even if P.Q is visible at the place where X is declared. 10.a

{*AI95-00408-01*} {*scope (of an attribute_)*} The scope of an attribute_definition_clause is identical to the scope of a declaration that would occur at the point of the attribute_definition_clause. 10.1/2

{*immediate scope (of (a view of) an entity)*} The immediate scope of a declaration is also the immediate scope of the entity or view declared by the declaration. {*scope (of (a view of) an entity)*} Similarly, the scope of a declaration is also the scope of the entity or view declared by the declaration. 11

Ramification: The rule for immediate scope implies the following: 11.a

- If the declaration is that of a library unit, then the immediate scope includes the declarative region of the declaration itself, but not other places, unless they are within the scope of a with_clause that mentions the library unit. 11.b

  It is necessary to attach the semantics of with_clauses to [immediate] scopes (as opposed to visibility), in order for various rules to work properly. A library unit should hide a homographic implicit declaration that appears in its parent, but only within the scope of a with_clause that mentions the library unit. Otherwise, we would violate the "legality determinable via semantic dependences" rule of 10, “Program Structure and Compilation Issues”. The declaration of a library unit should be allowed to be a homograph of an explicit declaration in its parent's body, so long as that body does not mention the library unit in a with_clause. 11.c

  This means that one cannot denote the declaration of the library unit, but one might still be able to denote the library unit via another view. 11.d

  A with_clause does not make the declaration of a library unit visible; the lack of a with_clause prevents it from being visible. Even if a library unit is mentioned in a with_clause, its declaration can still be hidden. 11.e

- The completion of the declaration of a library unit (assuming that's also a declaration) is not visible, neither directly nor by selection, outside that completion. 11.f

- The immediate scope of a declaration immediately within the body of a library unit does not include any child of that library unit. 11.g

  This is needed to prevent children from looking inside their parent's body. The children are in the declarative region of the parent, and they might be after the parent's body. Therefore, the scope of a declaration that occurs immediately within the body might include some children. 11.h

NOTES

4 There are notations for denoting visible declarations that are not directly visible. For example, parameter_specifications are in the visible part of a subprogram_declaration so that they can be used in named-notation calls appearing outside the called subprogram. For another example, declarations of the visible part of a package can be denoted by expanded names appearing outside the package, and can be made directly visible by a use_clause. 12

Ramification: {*AI95-00114-01*} There are some obscure cases involving generics in which there is no such notation. See Section 12. 12.a/2

Extensions to Ada 83

{*extensions to Ada 83*} The fact that the immediate scope of an overloadable declaration does not include its profile is new to Ada 95. It replaces RM83-8.3(16), which said that within a subprogram specification and within the formal part of an entry declaration or accept statement, all declarations with the same designator as the subprogram or entry were hidden from all visibility. The RM83-8.3(16) rule seemed to be overkill, and created both implementation difficulties and unnecessary semantic complexity. 12.b

Wording Changes from Ada 83

We no longer need to talk about the scope of notations, identifiers, character_literals, and operator_symbols. 12.c

The notion of "visible part" has been extended in Ada 95. The syntax of task and protected units now allows private parts, thus requiring us to be able to talk about the visible part as well. It was necessary to extend the concept to subprograms and to generic units, in order for the visibility rules related to child library units to work properly. It was necessary to define the concept separately for generic formal packages, since their visible part is slightly different from that of a normal package. Extending the concept to composite types made the definition of scope slightly simpler. We define visible part for some things elsewhere, since it makes a big difference to the user for those things. For composite types and subprograms, however, the concept is used only in arcane visibility rules, so we localize it to this clause. 12.d

12.e In Ada 83, the semantics of with_clauses was described in terms of visibility. It is now described in terms of [immediate] scope.

12.f We have clarified that the following is illegal (where Q and R are library units):

12.g
```
package Q is
    I : Integer := 0;
end Q;
```

12.h
```
package R is
    package X renames Standard;
    X.Q.I := 17; -- Illegal!
end R;
```

12.i even though Q is declared in the declarative region of Standard, because R does not mention Q in a with_clause.

Wording Changes from Ada 95

12.j/2 {*AI95-00408-01*} The scope of an attribute_definition_clause is defined so that it can be used to define the visibility of such a clause, so *that* can be used by the stream attribute availability rules (see 13.13.2).

8.3 Visibility

1 [{*visibility rules*} The *visibility rules*, given below, determine which declarations are visible and directly visible at each place within a program. The visibility rules apply to both explicit and implicit declarations.]

Static Semantics

2 {*visibility (direct)*} {*directly visible*} {*directly visible*} A declaration is defined to be *directly visible* at places where a name consisting of only an identifier or operator_symbol is sufficient to denote the declaration; that is, no selected_component notation or special context (such as preceding => in a named association) is necessary to denote the declaration. {*visible*} A declaration is defined to be *visible* wherever it is directly visible, as well as at other places where some name (such as a selected_component) can denote the declaration.

3 The syntactic category direct_name is used to indicate contexts where direct visibility is required. The syntactic category selector_name is used to indicate contexts where visibility, but not direct visibility, is required.

4 {*visibility (immediate)*} {*visibility (use clause)*} There are two kinds of direct visibility: *immediate visibility* and *use-visibility*. {*immediately visible*} A declaration is immediately visible at a place if it is directly visible because the place is within its immediate scope. {*use-visible*} A declaration is use-visible if it is directly visible because of a use_clause (see 8.4). Both conditions can apply.

5 {*hiding*} A declaration can be *hidden*, either from direct visibility, or from all visibility, within certain parts of its scope. {*hidden from all visibility*} Where *hidden from all visibility*, it is not visible at all (neither using a direct_name nor a selector_name). {*hidden from direct visibility*} Where *hidden from direct visibility*, only direct visibility is lost; visibility using a selector_name is still possible.

6 [{*overloaded*} Two or more declarations are *overloaded* if they all have the same defining name and there is a place where they are all directly visible.]

6.a **Ramification:** Note that a name can have more than one possible interpretation even if it denotes a non-overloadable entity. For example, if there are two functions F that return records, both containing a component called C, then the name F.C has two possible interpretations, even though component declarations are not overloadable.

7 {*overloadable*} The declarations of callable entities [(including enumeration literals)] are *overloadable*[, meaning that overloading is allowed for them].

7.a **Ramification:** A generic_declaration is not overloadable within its own generic_formal_part. This follows from the rules about when a name denotes a current instance. See AI83-00286. This implies that within a generic_formal_part, outer declarations with the same defining name are hidden from direct visibility. It also implies that if a generic formal parameter has the same defining name as the generic itself, the formal parameter hides the generic from direct visibility.

8 {*homograph*} Two declarations are *homographs* if they have the same defining name, and, if both are overloadable, their profiles are type conformant. {*type conformance* [partial]} [An inner declaration hides any outer homograph from direct visibility.]

8.a/2 **Glossary entry:** {*Overriding operation*} An overriding operation is one that replaces an inherited primitive operation. Operations may be marked explicitly as overriding or not overriding.

9/1 {*8652/0025*} {*AI95-00044-01*} [Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other (see Legality Rules below).] {*override*} The only declarations that are {*overridable*} *overridable* are the implicit declarations for predefined operators and inherited primitive subprograms. A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases:

- 10/1 {*8652/0025*} {*AI95-00044-01*} A declaration that is not overridable overrides one that is overridable, [regardless of which declaration occurs first];

10.a/1 **Ramification:** {*8652/0025*} {*AI95-00044-01*} And regardless of whether the non-overridable declaration is overloadable or not. For example, statement_identifiers are covered by this rule.

10.b The “regardless of which declaration occurs first” is there because the explicit declaration could be a primitive subprogram of a partial view, and then the full view might inherit a homograph. We are saying that the explicit one wins (within its scope), even though the implicit one comes later.

10.c If the overriding declaration is also a subprogram, then it is a primitive subprogram.

10.d As explained in 7.3.1, “Private Operations”, some inherited primitive subprograms are never declared. Such subprograms cannot be overridden, although they can be reached by dispatching calls in the case of a tagged type.

- 11 The implicit declaration of an inherited operator overrides that of a predefined operator;

11.a **Ramification:** In a previous version of Ada 9X, we tried to avoid the notion of predefined operators, and say that they were inherited from some magical root type. However, this seemed like too much mechanism. Therefore, a type can have a predefined "+" as well as an inherited "+". The above rule says the inherited one wins.

11.b/2 {*AI95-00114-01*} The “regardless of which declaration occurs first” applies here as well, in the case where derived_type_definition in the visible part of a public library unit derives from a private type declared in the parent unit, and the full view of the parent type has additional predefined operators, as explained in 7.3.1, “Private Operations”. Those predefined operators can be overridden by inherited subprograms implicitly declared earlier.

- 12 An implicit declaration of an inherited subprogram overrides a previous implicit declaration of an inherited subprogram.

- 12.1/2 {*AI95-00251-01*} If two or more homographs are implicitly declared at the same place:

  - 12.2/2 {*AI95-00251-01*} If at least one is a subprogram that is neither a null procedure nor an abstract subprogram, and does not require overriding (see 3.9.3), then they override those that are null procedures, abstract subprograms, or require overriding. If more than one such homograph remains that is not thus overridden, then they are all hidden from all visibility.

  - 12.3/2 {*AI95-00251-01*} Otherwise (all are null procedures, abstract subprograms, or require overriding), then any null procedure overrides all abstract subprograms and all subprograms that require overriding; if more than one such homograph remains that is not thus overridden, then if they are all fully conformant with one another, one is chosen arbitrarily; if not, they are all hidden from all visibility. {*full conformance (required)*}

12.a/2 **Discussion:** In the case where the implementation arbitrarily chooses one overrider from among a group of inherited subprograms, users should not be able to determine which member was chosen, as the set of inherited subprograms which are chosen from must be fully conformant. This rule is needed in order to allow

12.b/2
```
package Outer is
   package P1 is
      type Ifc1 is interface;
      procedure Null_Procedure (X : Ifc1) is null;
      procedure Abstract_Subp  (X : Ifc1) is abstract;
   end P1;
```

12.c/2
```
         package P2 is
            type Ifc2 is interface;
            procedure Null_Procedure (X : Ifc2) is null;
            procedure Abstract_Subp  (X : Ifc2) is abstract;
         end P2;
```

12.d/2
```
         type T is abstract new P1.Ifc1 and P2.Ifc2 with null record;
      end Outer;
```

12.e/2 without requiring that T explicitly override any of its inherited operations.

12.f/2 Full conformance is required here, as we cannot allow the parameter names to differ. If they did differ, the routine which was selected for overriding could be determined by using named parameter notation in a call.

12.g/2 When the subprograms do not conform, we chose not to adopt the "use clause" rule which would make them all visible resulting in likely ambiguity. If we had used such a rule, any successful calls would be confusing; and the fact that there are no Beaujolais-like effect to worry about means we can consider other rules. The hidden-from-all-visibility homographs are still inherited by further derivations, which avoids order-of-declaration dependencies and other anomalies.

12.h/2 We have to be careful to not include arbitrary selection if the routines have real bodies. (This can happen in generics, see the example in the incompatibilities section below.) We don't want the ability to successfully call routines where the body executed depends on the compiler or a phase of the moon.

12.i/2 Note that if the type is concrete, abstract subprograms are inherited as subprograms that require overriding. We include functions that require overriding as well; these don't have real bodies, so they can use the more liberal rules.

13
- [For an implicit declaration of a primitive subprogram in a generic unit, there is a copy of this declaration in an instance.] However, a whole new set of primitive subprograms is implicitly declared for each type declared within the visible part of the instance. These new declarations occur immediately after the type declaration, and override the copied ones. [The copied ones can be called only from within the instance; the new ones can be called only from outside the instance, although for tagged types, the body of a new one can be executed by a call to an old one.]

13.a **Discussion:** In addition, this is also stated redundantly (again), and is repeated, in 12.3, "Generic Instantiation". The rationale for the rule is explained there.

14 {*visible*} {*hidden from all visibility* [distributed]} A declaration is visible within its scope, except where hidden from all visibility, as follows:

15
- {*hidden from all visibility (for overridden declaration)* [partial]} An overridden declaration is hidden from all visibility within the scope of the overriding declaration.

15.a **Ramification:** We have to talk about the scope of the overriding declaration, not its visibility, because it hides even when it is itself hidden.

15.b Note that the scope of an explicit subprogram_declaration does not start until after its profile.

16
- {*hidden from all visibility (within the declaration itself)* [partial]} A declaration is hidden from all visibility until the end of the declaration, except:

17
  - For a record type or record extension, the declaration is hidden from all visibility only until the reserved word **record**;

18/2
  - {*AI95-00345-01*} For a package_declaration, generic_package_declaration, or subprogram_body, the declaration is hidden from all visibility only until the reserved word **is** of the declaration;

18.a **Ramification:** We're talking about the **is** of the construct itself, here, not some random **is** that might appear in a generic_formal_part.

18.1/2
  - {*AI95-00345-01*} For a task declaration or protected declaration, the declaration is hidden from all visibility only until the reserved word **with** of the declaration if there is one, or the reserved word **is** of the declaration if there is no **with**.

18.b/2 **To be honest:** If there is neither a **with** nor **is**, then the exception does not apply and the name is hidden from all visibility until the end of the declaration. This oddity was inherited from Ada 95.

18.c/2 **Reason:** We need the "**with** or **is**" rule so that the visibility within an interface_list does not vary by construct. That would make it harder to complete private extensions and would complicate implementations.

- {*hidden from all visibility (for a declaration completed by a subsequent declaration)* [partial]} If the completion of a declaration is a declaration, then within the scope of the completion, the first declaration is hidden from all visibility. Similarly, a discriminant_specification or parameter_-specification is hidden within the scope of a corresponding discriminant_specification or parameter_specification of a corresponding completion, or of a corresponding accept_-statement. 19

  Ramification: This rule means, for example, that within the scope of a full_type_declaration that completes a private_type_declaration, the name of the type will denote the full_type_declaration, and therefore the full view of the type. On the other hand, if the completion is not a declaration, then it doesn't hide anything, and you can't denote it. 19.a

- {*AI95-00217-06*} {*AI95-00412-01*} {*hidden from all visibility (by lack of a with_clause)* [partial]} The declaration of a library unit (including a library_unit_renaming_declaration) is hidden from all visibility at places outside its declarative region that are not within the scope of a nonlimited_with_clause that mentions it. The limited view of a library package is hidden from all visibility at places that are not within the scope of a limited_with_clause that mentions it; in addition, the limited view is hidden from all visibility within the declarative region of the package, as well as within the scope of any nonlimited_with_clause that mentions the package. Where the declaration of the limited view of a package is visible, any name that denotes the package denotes the limited view, including those provided by a package renaming. 20/2

  Discussion: {*AI95-00217-06*} This is the rule that prevents with_clauses from being transitive; the [immediate] scope includes indirect semantic dependents. This rule also prevents the limited view of a package from being visible in the same place as the full view of the package, which prevents various ripple effects. 20.a/2

- {*AI95-00217-06*} {*AI95-00412-01*} [For each declaration or renaming of a generic unit as a child of some parent generic package, there is a corresponding declaration nested immediately within each instance of the parent.] Such a nested declaration is hidden from all visibility except at places that are within the scope of a with_clause that mentions the child. 20.1/2

{*directly visible*} {*immediately visible*} {*visibility (direct)*} {*visibility (immediate)*} A declaration with a defining_identifier or defining_operator_symbol is immediately visible [(and hence directly visible)] within its immediate scope {*hidden from direct visibility* [distributed]} except where hidden from direct visibility, as follows: 21

- {*hidden from direct visibility (by an inner homograph)* [partial]} A declaration is hidden from direct visibility within the immediate scope of a homograph of the declaration, if the homograph occurs within an inner declarative region; 22

- {*hidden from direct visibility (where hidden from all visibility)* [partial]} A declaration is also hidden from direct visibility where hidden from all visibility. 23

{*AI95-00195-01*} {*AI95-00408-01*} {*visible (attribute_* [partial]} An attribute_definition_clause is *visible* everywhere within its scope. 23.1/2

Name Resolution Rules

{*possible interpretation (for direct_names)* [partial]} A direct_name shall resolve to denote a directly visible declaration whose defining name is the same as the direct_name. {*possible interpretation (for selector_names)* [partial]} A selector_name shall resolve to denote a visible declaration whose defining name is the same as the selector_name. 24

Discussion: "The same as" has the obvious meaning here, so for +, the possible interpretations are declarations whose defining name is "+" (an operator_symbol). 24.a

These rules on visibility and direct visibility do not apply in a context_clause, a parent_unit_name, or a pragma that appears at the place of a compilation_unit. For those contexts, see the rules in 10.1.6, “Environment-Level Visibility Rules”. 25

25.a **Ramification:** Direct visibility is irrelevant for character_literals. In terms of overload resolution character_literals are similar to other literals, like **null** — see 4.2. For character_literals, there is no need to worry about hiding, since there is no way to declare homographs.

Legality Rules

26/2 {*8652/0025*} {*8652/0026*} {*AI95-00044-01*} {*AI95-00150-01*} {*AI95-00377-01*} A non-overridable declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the non-overridable declaration. In addition, a type extension is illegal if somewhere within its immediate scope it has two visible components with the same name. Similarly, the context_clause for a compilation unit is illegal if it mentions (in a with_clause) some library unit, and there is a homograph of the library unit that is visible at the place of the compilation unit, and the homograph and the mentioned library unit are both declared immediately within the same declarative region.{*generic contract issue* [partial]} These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance[; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant]. {*type conformance* [partial]}

26.a **Discussion:** Normally, these rules just mean you can't explicitly declare two homographs immediately within the same declarative region. The wording is designed to handle the following special cases:

26.b
- If the second declaration completes the first one, the second declaration is legal.

26.c
- If the body of a library unit contains an explicit homograph of a child of that same library unit, this is illegal only if the body mentions the child in its context_clause, or if some subunit mentions the child. Here's an example:

26.d
```
package P is
end P;
```

26.e
```
package P.Q is
end P.Q;
```

26.f
```
package body P is
    Q : Integer; -- OK; we cannot see package P.Q here.
    procedure Sub is separate;
end P;
```

26.g
```
with P.Q;
separate(P)
procedure Sub is -- Illegal.
begin
    null;
end Sub;
```

26.h If package body P said "**with** P.Q;", then it would be illegal to declare the homograph Q: Integer. But it does not, so the body of P is OK. However, the subunit would be able to see both P.Q's, and is therefore illegal.

26.i A previous version of Ada 9X allowed the subunit, and said that references to P.Q would tend to be ambiguous. However, that was a bad idea, because it requires overload resolution to resolve references to directly visible non-overloadable homographs, which is something compilers have never before been required to do.

26.i.1/1
- {*8652/0026*} {*8652/0102*} {*AI95-00150-01*} {*AI95-00157-01*} If a type extension contains a component with the same name as a component in an ancestor type, there must be no place where both components are visible. For instance:

26.i.2/1
```
package A is
   type T is tagged private;
   package B is
      type NT is new T with record
         I: Integer; -- Illegal because T.I is visible in the body.
      end record; -- T.I is not visible here.
   end B;
private
   type T is tagged record
      I: Integer; -- Illegal because T.I is visible in the body.
   end record;
end A;
```

```
{AI95-00114-01} package body A is                                         26.i.3/2
   package body B is
      -- T.I becomes visible here.
   end B;
end A;
```

```
package A.C is                                                              26.i.4/1
   type NT2 is new A.T with record
      I: Integer; -- Illegal because T.I is visible in the private part.
   end record; -- T.I is not visible here.
private
    -- T.I is visible here.
end A.C;
```

```
with A;                                                                     26.i.5/1
package D is
   type NT3 is new A.T with record
      I: Integer; -- Legal because T.I is never visible in this package.
   end record;
end D;
```

```
with D;                                                                     26.i.6/1
package A.E is
   type NT4 is new D.NT3 with null record;
   X : NT4;
   I1 : Integer := X.I;        -- D.NT3.I
   I2 : Integer := D.NT3(X).I; -- D.NT3.I
   I3 : Integer := A.T(X).I;   -- A.T.I
end A.E;
```

26.i.7/1 {*8652/0102*} {*AI95-00157-01*} D.NT3 can have a component I because the component I of the parent type is never visible. The parent component exists, of course, but is never declared for the type D.NT3. In the child package A.E, the component I of A.T is visible, but that does not change the fact that the A.T.I component was never declared for type D.NT3. Thus, A.E.NT4 does not (visibly) inherit the component I from A.T, while it does inherit the component I from D.NT3. Of course, both components exist, and can be accessed by a type conversion as shown above. This behavior stems from the fact that every characteristic of a type (including components) must be declared somewhere in the innermost declarative region containing the type — if the characteristic is never visible in that declarative region, it is never declared. Therefore, such characteristics do not suddenly become available even if they are in fact visible in some other scope. See 7.3.1 for more on the rules.

- 26.i.8/2 {*AI95-00377-01*} It is illegal to mention both an explicit child of an instance, and a child of the generic from which the instance was instantiated. This is easier to understand with an example:

```
generic                                                                     26.i.9/2
package G1 is
end G1;
```

```
generic                                                                     26.i.10/2
package G1.G2 is
end G1.G2;
```

```
with G1;                                                                    26.i.11/2
package I1 is new G1;
```

```
package I1.G2 renames ...                                                   26.i.12/2
```

```
with G1.G2;                                                                 26.i.13/2
with I1.G2;             -- Illegal
package Bad is ...
```

26.i.14/2 The context clause for Bad is illegal as I1 has an implicit declaration of I1.G2 based on the generic child G1.G2, as well as the mention of the explicit child I1.G2. As in the previous cases, this is illegal only if the context clause makes both children visible; the explicit child can be mentioned as long as the generic child is not (and vice-versa).

26.j Note that we need to be careful which things we make "hidden from all visibility" versus which things we make simply illegal for names to denote. The distinction is subtle. The rules that disallow names denoting components within a type declaration (see 3.7) do not make the components invisible at those places, so that the above rule makes components with the same name illegal. The same is true for the rule that disallows names denoting formal parameters within a formal_part (see 6.1).

26.k **Discussion:** The part about instances is from AI83-00012. The reason it says “overloadable declarations” is because we don't want it to apply to type extensions that appear in an instance; components are not overloadable.

NOTES

27 5 Visibility for compilation units follows from the definition of the environment in 10.1.4, except that it is necessary to apply a with_clause to obtain visibility to a library_unit_declaration or library_unit_renaming_declaration.

28 6 In addition to the visibility rules given above, the meaning of the occurrence of a direct_name or selector_name at a given place in the text can depend on the overloading rules (see 8.6).

29 7 Not all contexts where an identifier, character_literal, or operator_symbol are allowed require visibility of a corresponding declaration. Contexts where visibility is not required are identified by using one of these three syntactic categories directly in a syntax rule, rather than using direct_name or selector_name.

29.a **Ramification:** An identifier, character_literal or operator_symbol that occurs in one of the following contexts is not required to denote a visible or directly visible declaration:

29.b 1. A defining name.

29.c 2. The identifiers or operator_symbol that appear after the reserved word **end** in a proper_body. Similarly for “**end loop**”, etc.

29.d 3. An attribute_designator.

29.e 4. A pragma identifier.

29.f 5. A *pragma_argument_*identifier.

29.g 6. An identifier specific to a pragma used in a pragma argument.

29.h The visibility rules have nothing to do with the above cases; the meanings of such things are defined elsewhere. Reserved words are not identifiers; the visibility rules don't apply to them either.

29.i Because of the way we have defined "declaration", it is possible for a usage name to denote a subprogram_body, either within that body, or (for a non-library unit) after it (since the body hides the corresponding declaration, if any). Other bodies do not work that way. Completions of type_declarations and deferred constant declarations do work that way. Accept_statements are never denoted, although the parameter_specifications in their profiles can be.

29.j The scope of a subprogram does not start until after its profile. Thus, the following is legal:

29.k
```
X : constant Integer := 17;
...
package P is
    procedure X(Y : in Integer := X);
end P;
```

29.l The body of the subprogram will probably be illegal, however, since the constant X will be hidden by then.

29.m The rule is different for generic subprograms, since they are not overloadable; the following is illegal:

29.n
```
X : constant Integer := 17;
package P is
    generic
      Z : Integer := X; -- Illegal!
    procedure X(Y : in Integer := X); -- Illegal!
end P;
```

29.o The constant X is hidden from direct visibility by the generic declaration.

Extensions to Ada 83

29.p {*extensions to Ada 83*} Declarations with the same defining name as that of a subprogram or entry being defined are nevertheless visible within the subprogram specification or entry declaration.

Wording Changes from Ada 83

29.q The term “visible by selection” is no longer defined. We use the terms “directly visible” and “visible” (among other things). There are only two regions of text that are of interest, here: the region in which a declaration is visible, and the region in which it is directly visible.

29.r Visibility is defined only for declarations.

Incompatibilities With Ada 95

29.s/2 {*AI95-00251-01*} {*incompatibilities with Ada 95*} Added rules to handle the inheritance and overriding of multiple homographs for a single type declaration, in order to support multiple inheritance from interfaces. The new rules are intended to be compatible with the existing rules so that programs that do not use interfaces do not change their legality. However, there is a very rare case where this is not true:

```
generic                                                        29.t/2
   type T1 is private;
   type T2 is private;
package G is
   type T is null record;
   procedure P (X : T; Y : T1);
   procedure P (X : T; Z : T2);
end G;]
```

```
package I is new G (Integer, Integer); -- Exports homographs of P.   29.u/2
```

```
type D is new I.T; -- Both Ps are inherited.                         29.v/2
```

```
Obj : D;                                                             29.w/2
```

```
P (Obj, Z => 10); -- Legal in Ada 95, illegal in Ada 2005.           29.x/2
```

The call to P would resolve in Ada 95 by using the parameter name, while the procedures P would be hidden from all visibility in Ada 2005 and thus would not resolve. This case doesn't seem worth making the rules any more complex than they already are. 29.y/2

{*AI95-00377-01*} **Amendment Correction:** A with_clause is illegal if it would create a homograph of an implicitly declared generic child (see 10.1.1). An Ada 95 compiler could have allowed this, but which unit of the two units involved would be denoted wasn't specified, so any successful use isn't portable. Removing one of the two with_clauses involved will fix the problem. 29.z/2

Wording Changes from Ada 95

{*8652/0025*} {*AI95-00044-01*} **Corrigendum:** Clarified the overriding rules so that "/=" and statement_identifiers are covered. 29.aa/2

{*8652/0026*} {*AI95-00150-01*} **Corrigendum:** Clarified that is it never possible for two components with the same name to be visible; any such program is illegal. 29.bb/2

{*AI95-00195-01*} {*AI95-00408-01*} The visibility of an attribute_definition_clause is defined so that it can be used by the stream attribute availability rules (see 13.13.2). 29.cc/2

{*AI95-00217-06*} The visibility of a limited view of a library package is defined (see 10.1.1). 29.dd/2

8.3.1 Overriding Indicators

{*AI95-00218-03*} An overriding_indicator is used to declare that an operation is intended to override (or not override) an inherited operation. 1/2

Syntax

{*AI95-00218-03*} overriding_indicator ::= [**not**] **overriding** 2/2

Legality Rules

{*AI95-00218-03*} {*AI95-00348-01*} {*AI95-00397-01*} If an abstract_subprogram_declaration, null_procedure_declaration, subprogram_body, subprogram_body_stub, subprogram_renaming_declaration, generic_instantiation of a subprogram, or subprogram_declaration other than a protected subprogram has an overriding_indicator, then: 3/2

- the operation shall be a primitive operation for some type; 4/2
- if the overriding_indicator is **overriding**, then the operation shall override a homograph at the place of the declaration or body; 5/2
- if the overriding_indicator is **not overriding**, then the operation shall not override any homograph (at any place). 6/2

{*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply, these rules also apply in the private part of an instance of a generic unit. 7/2

Discussion: The **overriding** and **not overriding** rules differ slightly. For **overriding**, we want the indicator to reflect the overriding state at the place of the declaration; otherwise the indicator would be “lying”. Whether a homograph is implicitly declared after the declaration (see 7.3.1 to see how this can happen) has no impact on this check. However, **not overriding** is different; “lying” would happen if a homograph declared later actually is overriding. So, we require 7.a/2 this check to take into account later overridings. That can be implemented either by looking ahead, or by rechecking when additional operations are declared.

7.b/2 The "no lying" rules are needed to prevent a subprogram_declaration and subprogram_body from having contradictory overriding_indicators.

NOTES

8/2 8 {*AI95-00397-01*} Rules for overriding_indicators of task and protected entries and of protected subprograms are found in 9.5.2 and 9.4, respectively.

Examples

9/2 {*AI95-00433-01*} The use of overriding_indicators allows the detection of errors at compile-time that otherwise might not be detected at all. For instance, we might declare a security queue derived from the Queue interface of 3.9.4 as:

```
10/2    type Security_Queue is new Queue with record ...;

11/2    overriding
        procedure Append(Q : in out Security_Queue; Person : in Person_Name);

12/2    overriding
        procedure Remove_First(Q : in out Security_Queue; Person : in Person_Name);

13/2    overriding
        function Cur_Count(Q : in Security_Queue) return Natural;

14/2    overriding
        function Max_Count(Q : in Security_Queue) return Natural;

15/2    not overriding
        procedure Arrest(Q : in out Security_Queue; Person : in Person_Name);
```

16/2 The first four subprogram declarations guarantee that these subprograms will override the four subprograms inherited from the Queue interface. A misspelling in one of these subprograms will be detected by the implementation. Conversely, the declaration of Arrest guarantees that this is a new operation.

16.a/2 **Discussion:** In this case, the subprograms are abstract, so misspellings will get detected anyway. But for other subprograms (especially when deriving from concrete types), the error might never be detected, and a body other than the one the programmer intended might be executed without warning. Thus our new motto: "Overriding indicators — don't derive a type without them!"

Extensions to Ada 95

16.b/2 {*AI95-00218-03*} {*extensions to Ada 95*} Overriding_indicators are new. These let the programmer state her overriding intentions to the compiler; if the compiler disagrees, an error will be produced rather than a hard to find bug.

8.4 Use Clauses

1 [A use_package_clause achieves direct visibility of declarations that appear in the visible part of a package; a use_type_clause achieves direct visibility of the primitive operators of a type.]

Language Design Principles

1.a {*equivalence of use_clauses and selected_components*} If and only if the visibility rules allow P.A, "**use** P;" should make A directly visible (barring name conflicts). This means, for example, that child library units, and generic formals of a formal package whose formal_package_actual_part is (<>), should be made visible by a use_clause for the appropriate package.

1.b {*Beaujolais effect*} The rules for use_clauses were carefully constructed to avoid so-called *Beaujolais* effects, where the addition or removal of a single use_clause, or a single declaration in a "use"d package, would change the meaning of a program from one legal interpretation to another.

Syntax

2 use_clause ::= use_package_clause | use_type_clause

use_package_clause ::= **use** *package*_name {, *package*_name}; 3

use_type_clause ::= **use type** subtype_mark {, subtype_mark}; 4

Legality Rules

{*AI95-00217-06*} A *package*_name of a use_package_clause shall denote a nonlimited view of a package. 5/2

Ramification: This includes formal packages. 5.a

Static Semantics

{*scope (of a use_clause)*} For each use_clause, there is a certain region of text called the *scope* of the use_clause. For a use_clause within a context_clause of a library_unit_declaration or library_unit_renaming_declaration, the scope is the entire declarative region of the declaration. For a use_clause within a context_clause of a body, the scope is the entire body [and any subunits (including multiply nested subunits). The scope does not include context_clauses themselves.] 6

For a use_clause immediately within a declarative region, the scope is the portion of the declarative region starting just after the use_clause and extending to the end of the declarative region. However, the scope of a use_clause in the private part of a library unit does not include the visible part of any public descendant of that library unit. 7

Reason: The exception echoes the similar exception for “immediate scope (of a declaration)” (see 8.2). It makes use_clauses work like this: 7.a

7.b
```
package P is
    type T is range 1..10;
end P;
```

7.c
```
with P;
package Parent is
private
    use P;
    X : T;
end Parent;
```

7.d
```
package Parent.Child is
    Y : T; -- Illegal!
    Z : P.T;
private
    W : T;
end Parent.Child;
```

The declaration of Y is illegal because the scope of the “**use** P” does not include that place, so T is not directly visible there. The declarations of X, Z, and W are legal. 7.e

{*AI95-00217-06*} A package is *named* in a use_package_clause if it is denoted by a *package*_name of that clause. A type is *named* in a use_type_clause if it is determined by a subtype_mark of that clause.{*named (in a use clause)*} 7.1/2

{*AI95-00217-06*} {*potentially use-visible*} For each package named in a use_package_clause whose scope encloses a place, each declaration that occurs immediately within the declarative region of the package is *potentially use-visible* at this place if the declaration is visible at this place. For each type *T* or *T*'Class named in a use_type_clause whose scope encloses a place, the declaration of each primitive operator of type *T* is potentially use-visible at this place if its declaration is visible at this place. 8/2

Ramification: Primitive subprograms whose defining name is an identifier are *not* made potentially visible by a use_type_clause. A use_type_clause is only for operators. 8.a

The semantics described here should be similar to the semantics for expanded names given in 4.1.3, “Selected Components” so as to achieve the effect requested by the “principle of equivalence of use_clauses and selected_components.” Thus, child library units and generic formal parameters of a formal package are potentially use-visible when their enclosing package is use'd. 8.b

8.c The "visible at that place" part implies that applying a use_clause to a parent unit does not make all of its children use-visible — only those that have been made visible by a with_clause. It also implies that we don't have to worry about hiding in the definition of "directly visible" — a declaration cannot be use-visible unless it is visible.

8.d Note that "**use type** T'Class;" is equivalent to "**use type** T;", which helps avoid breaking the generic contract model.

9 {*use-visible*} {*visibility (use clause)*} A declaration is *use-visible* if it is potentially use-visible, except in these naming-conflict cases:

10 • A potentially use-visible declaration is not use-visible if the place considered is within the immediate scope of a homograph of the declaration.

11 • Potentially use-visible declarations that have the same identifier are not use-visible unless each of them is an overloadable declaration.

11.a **Ramification:** Overloadable declarations don't cancel each other out, even if they are homographs, though if they are not distinguishable by formal parameter names or the presence or absence of default_expressions, any use will be ambiguous. We only mention identifiers here, because declarations named by operator_symbols are always overloadable, and hence never cancel each other. Direct visibility is irrelevant for character_literals.

Dynamic Semantics

12 {*elaboration (use_clause)* [partial]} The elaboration of a use_clause has no effect.

Examples

13 *Example of a use clause in a context clause:*

14
```
with Ada.Calendar; use Ada;
```

15 *Example of a use type clause:*

16
```
use type Rational_Numbers.Rational; -- see 7.1
Two_Thirds: Rational_Numbers.Rational := 2/3;
```

16.a **Ramification:** In “**use** X, Y;”, Y cannot refer to something made visible by the “**use**” of X. Thus, it's not (quite) equivalent to “**use** X; **use** Y;”.

16.b If a given declaration is already immediately visible, then a use_clause that makes it potentially use-visible has no effect. Therefore, a use_type_clause for a type whose declaration appears in a place other than the visible part of a package has no effect; it cannot make a declaration use-visible unless that declaration is already immediately visible.

16.c "**Use type** S1;" and "**use type** S2;" are equivalent if S1 and S2 are both subtypes of the same type. In particular, "**use type** S;" and "**use type** S'Base;" are equivalent.

16.d **Reason:** We considered adding a rule that prevented several declarations of views of the same entity that all have the same semantics from cancelling each other out. For example, if a (possibly implicit) subprogram_declaration for "+" is potentially use-visible, and a fully conformant renaming of it is also potentially use-visible, then they (annoyingly) cancel each other out; neither one is use-visible. The considered rule would have made just one of them use-visible. We gave up on this idea due to the complexity of the rule. It would have had to account for both overloadable and non-overloadable renaming_declarations, the case where the rule should apply only to some subset of the declarations with the same defining name, and the case of subtype_declarations (since they are claimed to be sufficient for renaming of subtypes).

Extensions to Ada 83

16.e {*extensions to Ada 83*} The use_type_clause is new to Ada 95.

Wording Changes from Ada 83

16.f The phrase “omitting from this set any packages that enclose this place” is no longer necessary to avoid making something visible outside its scope, because we explicitly state that the declaration has to be visible in order to be potentially use-visible.

Wording Changes from Ada 95

16.g/2 {*AI95-00217-06*} Limited views of packages are not allowed in use clauses. Defined *named in a use clause* for use in other limited view rules (see 10.1.2).

8.5 Renaming Declarations

1 [A renaming_declaration declares another name for an entity, such as an object, exception, package, subprogram, entry, or generic unit. Alternatively, a subprogram_renaming_declaration can be the completion of a previous subprogram_declaration.]

1.a.1/2 **Glossary entry:** {*Renaming*} A renaming_declaration is a declaration that does not define a new entity, but instead defines a view of an existing entity.

Syntax

2
renaming_declaration ::=
  object_renaming_declaration
 | exception_renaming_declaration
 | package_renaming_declaration
 | subprogram_renaming_declaration
 | generic_renaming_declaration

Dynamic Semantics

3 {*elaboration (renaming_declaration)* [partial]} The elaboration of a renaming_declaration evaluates the name that follows the reserved word **renames** and thereby determines the view and entity denoted by this name {*renamed view*} {*renamed entity*} (the *renamed view* and *renamed entity*). [A name that denotes the renaming_declaration denotes (a new view of) the renamed entity.]

NOTES

4 9 Renaming may be used to resolve name conflicts and to act as a shorthand. Renaming with a different identifier or operator_symbol does not hide the old name; the new name and the old name need not be visible at the same places.

5 10 A task or protected object that is declared by an explicit object_declaration can be renamed as an object. However, a single task or protected object cannot be renamed since the corresponding type is anonymous (meaning it has no nameable subtypes). For similar reasons, an object of an anonymous array or access type cannot be renamed.

6 11 A subtype defined without any additional constraint can be used to achieve the effect of renaming another subtype (including a task or protected subtype) as in

7
```
subtype Mode is Ada.Text_IO.File_Mode;
```

Wording Changes from Ada 83

7.a The second sentence of RM83-8.5(3), “At any point where a renaming declaration is visible, the identifier, or operator symbol of this declaration denotes the renamed entity.” is incorrect. It doesn't say directly visible. Also, such an identifier might resolve to something else.

7.b The verbiage about renamings being legal “only if exactly one...”, which appears in RM83-8.5(4) (for objects) and RM83-8.5(7) (for subprograms) is removed, because it follows from the normal rules about overload resolution. For language lawyers, these facts are obvious; for programmers, they are irrelevant, since failing these tests is highly unlikely.

8.5.1 Object Renaming Declarations

1 [An object_renaming_declaration is used to rename an object.]

Syntax

2/2 {*AI95-00230-01*} {*AI95-00423-01*} object_renaming_declaration ::=
  defining_identifier : [null_exclusion] subtype_mark **renames** *object*_name;
 | defining_identifier : access_definition **renames** *object*_name;

Name Resolution Rules

3/2 {*AI95-00230-01*} {*AI95-00254-01*} {*AI95-00409-01*} The type of the *object*_name shall resolve to the type determined by the subtype_mark, or in the case where the type is defined by an access_definition, to an anonymous access type. If the anonymous access type is an access-to-object type, the type of the *object*_name shall have the same designated type as that of the access_definition. If the anonymous access type is an access-to-subprogram type, the type of the *object*_name shall have a designated profile that is type conformant with that of the access_definition.

3.a **Reason:** A previous version of Ada 9X used the usual “expected type” wording:
“The expected type for the *object*_name is that determined by the subtype_mark.”
We changed it so that this would be illegal:

3.b
```
X: T;
Y: T'Class renames X; -- Illegal!
```

3.c When the above was legal, it was unclear whether Y was of type T or T'Class. Note that we still allow this:

3.d
```
Z: T'Class := ...;
W: T renames F(Z);
```

3.e where F is a function with a controlling parameter and result. This is admittedly a bit odd.

3.f Note that the matching rule for generic formal parameters of mode **in out** was changed to keep it consistent with the rule for renaming. That makes the rule different for **in** vs. **in out**.

Legality Rules

4 The renamed entity shall be an object.

4.1/2 {*AI95-00231-01*} {*AI95-00409-01*} In the case where the type is defined by an access_definition, the type of the renamed object and the type defined by the access_definition:

- 4.2/2 {*AI95-00231-01*} {*AI95-00409-01*} shall both be access-to-object types with statically matching designated subtypes and with both or neither being access-to-constant types; or {*statically matching (required)* [partial]}
- 4.3/2 {*AI95-00409-01*} shall both be access-to-subprogram types with subtype conformant designated profiles. {*subtype conformance (required)*}

4.4/2 {*AI95-00423-01*} For an object_renaming_declaration with a null_exclusion or an access_definition that has a null_exclusion:

- 4.5/2 if the *object*_name denotes a generic formal object of a generic unit *G*, and the object_renaming_declaration occurs within the body of *G* or within the body of a generic unit declared within the declarative region of *G*, then the declaration of the formal object of *G* shall have a null_exclusion;
- 4.6/2 otherwise, the subtype of the *object*_name shall exclude null. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

4.a/2 **Reason:** This rule prevents “lying”. **Null** must never be the value of an object with an explicit null_exclusion. The first bullet is an assume-the-worst rule which prevents trouble in one obscure case:

4.b/2
```
type Acc_I is access Integer;
subtype Acc_NN_I is not null Acc_I;
Obj : Acc_I := null;
```

4.c/2
```
generic
   B : in out Acc_NN_I;
package Gen is
   ...
end Gen;
```

4.d/2
```
package body Gen is
   D : not null Acc_I renames B;
end Gen;
```

4.e/2
```
package Inst is new Gen (B => Obj);
```

Without the first bullet rule, D would be legal, and contain the value **null**, because the rule about lying is satisfied for generic matching (Obj matches B; B does not explicitly state **not null**), Legality Rules are not rechecked in the body of any instance, and the template passes the lying rule as well. The rule is so complex because it has to apply to formals used in bodies of child generics as well as in the bodies of generics. 4.f/2

{*8652/0017*} {*AI95-00184-01*} {*AI95-00363-01*} The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is constrained by its initial value. A slice of an array shall not be renamed if this restriction disallows renaming of the array. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit. These rules also apply for a renaming that appears in the body of a generic unit, with the additional requirement that even if the nominal subtype of the variable is indefinite, its type shall not be a descendant of an untagged generic formal derived type. 5/2

Reason: This prevents renaming of subcomponents that might disappear, which might leave dangling references. Similar restrictions exist for the Access attribute. 5.a

{*8652/0017*} {*AI95-00184-01*} The "recheck on instantiation" and "assume-the-worst in the body" restrictions on generics are necessary to avoid renaming of components which could disappear even when the nominal subtype would prevent the problem: 5.a.1/1

5.a.2/1
```
type T1 (D1 : Boolean) is
   record
      case D1 is
         when False =>
            C1 : Integer;
         when True =>
            null;
         end case;
      end record;
```

5.a.3/1
```
generic
   type F is new T1;
   X : in out F;
package G is
   C1_Ren : Integer renames X.C1;
end G;
```

5.a.4/1
```
type T2 (D2 : Boolean := False) is new T1 (D1 => D2);

Y : T2;

package I is new G (T2, Y);

Y := (D1 => True); -- Oops!  What happened to I.C1_Ren?
```

Implementation Note: Note that if an implementation chooses to deallocate-then-reallocate on assignment_statements assigning to unconstrained definite objects, then it cannot represent renamings and access values as simple addresses, because the above rule does not apply to all components of such an object. 5.b

Ramification: If it is a generic formal object, then the assume-the-best or assume-the-worst rules are applied as appropriate. 5.c

To be honest: {*AI95-00363-01*} If renamed entity is a subcomponent that depends on discriminants, and the subcomponent is a dereference of a general access type whose designated type is unconstrained and whose discriminants have defaults, the renaming is illegal. Such a general access type can designate an unconstrained (stack) object. Since such a type might not designate an object constrained by its initial value, the renaming is illegal — the rule says "is" constrained by its initial value, not "might be" constrained by its initial value. No other interpretation makes sense, as we can't have legality depending on something (which object is designated) that is not known at compile-time, and we surely can't allow this for unconstrained objects. The wording of the rule should be much clearer on this point, but this was discovered after the completion of Amendment 1 when it was too late to fix it. 5.d/2

Static Semantics

{*AI95-00230-01*} {*AI95-00409-01*} An object_renaming_declaration declares a new view [of the renamed object] whose properties are identical to those of the renamed view. [Thus, the properties of the renamed object are not affected by the renaming_declaration. In particular, its value and whether or not it is a constant are unaffected; similarly, the null exclusion or constraints that apply to an object are not 6/2 affected by renaming (any constraint implied by the subtype_mark or access_definition of the object_renaming_declaration is ignored).]

6.a **Discussion:** Because the constraints are ignored, it is a good idea to use the nominal subtype of the renamed object when writing an object_renaming_declaration.

6.b/2 {*AI95-00409-01*} If no null_exclusion is given in the renaming, the object may or may not exclude null. This is similar to the way that constraints need not match, and **constant** is not specified. The renaming defines a view of the renamed entity, inheriting the original properties.

Examples

7 *Example of renaming an object:*

8
```
declare
   L : Person renames Leftmost_Person; -- see 3.10.1
begin
   L.Age := L.Age + 1;
end;
```

Wording Changes from Ada 83

8.a The phrase “subtype ... as defined in a corresponding object declaration, component declaration, or component subtype indication,” from RM83-8.5(5), is incorrect in Ada 95; therefore we removed it. It is incorrect in the case of an object with an indefinite unconstrained nominal subtype.

Incompatibilities With Ada 95

8.b/2 {*AI95-00363-01*} {*incompatibilities with Ada 95*} Aliased variables are not necessarily constrained in Ada 2005 (see 3.6). Therefore, a subcomponent of an aliased variable may disappear or change shape, and renaming such a subcomponent thus is illegal, while the same operation would have been legal in Ada 95. Note that most allocated objects are still constrained by their initial value (see 4.8), and thus have no change in the legality of renaming for them. For example, using the type T2 of the previous example:

8.c/2
```
AT2 : aliased T2;
C1_Ren : Integer renames AT2.C1; -- Illegal in Ada 2005, legal in Ada 95
AT2 := (D1 => True);             -- Raised Constraint_Error in Ada 95,
                                 -- but does not in Ada 2005, so C1_Ren becomes
                                 -- invalid when this is assigned.
```

Extensions to Ada 95

8.d/2 {*AI95-00230-01*} {*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00409-01*} {*extensions to Ada 95*} A renaming can have an anonymous access type. In that case, the accessibility of the renaming is that of the original object (accessibility is not lost as it is for a component or stand-alone object).

8.e/2 {*AI95-00231-01*} {*AI95-00423-01*} A renaming can have a null_exclusion; if so, the renamed object must also exclude null, so that the null_exclusion does not lie. On the other hand, if the renaming does not have a null_exclusion. it excludes null of the renamed object does.

Wording Changes from Ada 95

8.f/2 {*8652/0017*} {*AI95-00184-01*} **Corrigendum:** Fixed to forbid renamings of depends-on-discriminant components if the type *might* be definite.

8.5.2 Exception Renaming Declarations

1 [An exception_renaming_declaration is used to rename an exception.]

Syntax

2 exception_renaming_declaration ::= defining_identifier : **exception renames** *exception*_name;

Legality Rules

3 The renamed entity shall be an exception.

Static Semantics

An exception_renaming_declaration declares a new view [of the renamed exception]. 4

Examples

Example of renaming an exception: 5

```
EOF : exception renames Ada.IO_Exceptions.End_Error; -- see A.13
```
6

8.5.3 Package Renaming Declarations

[A package_renaming_declaration is used to rename a package.] 1

Syntax

package_renaming_declaration ::=
package defining_program_unit_name **renames** *package*_name; 2

Legality Rules

The renamed entity shall be a package. 3

{*AI95-00217-06*} {*AI95-00412-01*} If the *package*_name of a package_renaming_declaration denotes a limited view of a package *P*, then a name that denotes the package_renaming_declaration shall occur only within the immediate scope of the renaming or the scope of a with_clause that mentions the package *P* or, if *P* is a nested package, the innermost library package enclosing *P*. 3.1/2

Discussion: The use of a renaming that designates a limited view is restricted to locations where we know whether the view is limited or nonlimited (based on a with_clause). We don't want to make an implicit limited view, as those are not transitive like a regular view. Implementations should be able to see all limited views needed based on the context_clause. 3.a.1/2

Static Semantics

A package_renaming_declaration declares a new view [of the renamed package]. 4

{*AI95-00412-01*} [At places where the declaration of the limited view of the renamed package is visible, a name that denotes the package_renaming_declaration denotes a limited view of the package (see 10.1.1).] 4.1/2

Proof: This rule is found in 8.3, “Visibility”. 4.a.1/2

Examples

Example of renaming a package: 5

```
package TM renames Table_Manager;
```
6

Wording Changes from Ada 95

{*AI95-00217-06*} {*AI95-00412-01*} Uses of renamed limited views of packages can only be used within the scope of a with_clause for the renamed package. 6.a/2

8.5.4 Subprogram Renaming Declarations

A subprogram_renaming_declaration can serve as the completion of a subprogram_declaration; {*renaming-as-body*} such a renaming_declaration is called a *renaming-as-body*. {*renaming-as-declaration*} A subprogram_renaming_declaration that is not a completion is called a *renaming-as-declaration*[, and is used to rename a subprogram (possibly an enumeration literal) or an entry]. 1

Ramification: A renaming-as-body is a declaration, as defined in Section 3. 1.a

Syntax

2/2 {*AI95-00218-03*} subprogram_renaming_declaration ::=
  [overriding_indicator]
  subprogram_specification **renames** *callable_entity_*name;

Name Resolution Rules

3 {*expected profile (subprogram_renaming_declaration)* [partial]} The expected profile for the *callable_entity_*name is the profile given in the subprogram_specification.

Legality Rules

4 The profile of a renaming-as-declaration shall be mode-conformant with that of the renamed callable entity. {*mode conformance (required)*}

4.1/2 {*AI95-00423-01*} For a parameter or result subtype of the subprogram_specification that has an explicit null_exclusion:

4.2/2
- if the *callable_entity_*name denotes a generic formal subprogram of a generic unit *G*, and the subprogram_renaming_declaration occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of the generic unit *G*, then the corresponding parameter or result subtype of the formal subprogram of *G* shall have a null_exclusion;

4.3/2
- otherwise, the subtype of the corresponding parameter or result type of the renamed callable entity shall exclude null. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

4.a/2 **Reason:** This rule prevents “lying”. **Null** must never be the value of a parameter or result with an explicit null_exclusion. The first bullet is an assume-the-worst rule which prevents trouble in generic bodies (including bodies of child units) when the formal subtype excludes null implicitly.

5/1 {*8652/0027*} {*8652/0028*} {*AI95-00135-01*} {*AI95-00145-01*} The profile of a renaming-as-body shall conform fully to that of the declaration it completes. {*full conformance (required)*} If the renaming-as-body completes that declaration before the subprogram it declares is frozen, the profile shall be mode-conformant {*mode conformance (required)*} with that of the renamed callable entity and the subprogram it declares takes its convention from the renamed subprogram; otherwise, the profile shall be subtype-conformant with that of the renamed callable entity and the convention of the renamed subprogram shall not be Intrinsic. {*subtype conformance (required)*} A renaming-as-body is illegal if the declaration occurs before the subprogram whose declaration it completes is frozen, and the renaming renames the subprogram itself, through one or more subprogram renaming declarations, none of whose subprograms has been frozen.

5.a/1 **Reason:** The otherwise part of the second sentence is to allow an implementation of a renaming-as-body as a single jump instruction to the target subprogram. Among other things, this prevents a subprogram from being completed with a renaming of an entry. (In most cases, the target of the jump can be filled in at link time. In some cases, such as a renaming of a name like "A(I).**all**", an indirect jump is needed. Note that the name is evaluated at renaming time, not at call time.)

5.a.1/1 {*8652/0028*} {*AI95-00145-01*} The first part of the second sentence is intended to allow renaming-as-body of predefined operators before the subprogram_declaration is frozen. For some types (such as integer types), the parameter type for operators is the base type, and it would be very strange for

```
function Equal (A, B : in T) return Boolean;
function Equal (A, B : in T) return Boolean renames "=";
```

to be illegal. (Note that predefined operators cannot be renamed this way after the subprogram_declaration is frozen, as they have convention Intrinsic.)

5.b/1 The first sentence is the normal rule for completions of subprogram_declarations.

5.c **Ramification:** An entry_declaration, unlike a subprogram_declaration, cannot be completed with a renaming_-declaration. Nor can a generic_subprogram_declaration.

The syntax rules prevent a protected subprogram declaration from being completed by a renaming. This is fortunate, because it allows us to avoid worrying about whether the implicit protected object parameter of a protected operation is involved in the conformance rules. 5.d

Reason: {*8652/0027*} {*AI95-00135-01*} Circular renames before freezing is illegal, as the compiler would not be able to determine the convention of the subprogram. Other circular renames are handled below; see Bounded (Run-Time) Errors. 5.d.1/1

{*AI95-00228-01*} The *callable_entity*_name of a renaming shall not denote a subprogram that requires overriding (see 3.9.3). 5.1/2

Reason: {*AI95-00228-01*} Such a rename cannot be of the inherited subprogram (which requires overriding because it cannot be called), and thus cannot squirrel away a subprogram (see below). That would be confusing, so we make it illegal. The renaming is allowed after the overriding, as then the name will denote the overriding subprogram, not the inherited one. 5.d.2/2

{*AI95-00228-01*} The *callable_entity*_name of a renaming-as-body shall not denote an abstract subprogram. 5.2/2

Reason: {*AI95-00228-01*} Such a subprogram has no body, so it hardly can replace one in the program. 5.d.3/2

A name that denotes a formal parameter of the subprogram_specification is not allowed within the *callable_entity*_name. 6

Reason: This is to prevent things like this: 6.a

```
function F(X : Integer) return Integer renames Table(X).all;
```
6.b

A similar rule in 6.1 forbids things like this: 6.c

```
function F(X : Integer; Y : Integer := X) return Integer;
```
6.d

Static Semantics

A renaming-as-declaration declares a new view of the renamed entity. The profile of this new view takes its subtypes, parameter modes, and calling convention from the original profile of the callable entity, while taking the formal parameter names and default_expressions from the profile given in the subprogram_renaming_declaration. The new view is a function or procedure, never an entry. 7

To be honest: When renaming an entry as a procedure, the compile-time rules apply as if the new view is a procedure, but the run-time semantics of a call are that of an entry call. 7.a

Ramification: For example, it is illegal for the entry_call_statement of a timed_entry_call to call the new view. But what looks like a procedure call will do things like barrier waiting. 7.b

{*8652/0105*} {*AI95-00211-01*} {*AI95-00228-01*} All properties of the renamed entity are inherited by the new view unless otherwise stated by this International Standard. In particular, if the renamed entity is abstract, the new view also is abstract. 7.b.1/2

Dynamic Semantics

{*8652/0014*} {*AI95-00064-01*} For a call to a subprogram whose body is given as a renaming-as-body, the execution of the renaming-as-body is equivalent to the execution of a subprogram_body that simply calls the renamed subprogram with its formal parameters as the actual parameters and, if it is a function, returns the value of the call. 7.1/1

Ramification: This implies that the subprogram completed by the renaming-as-body has its own elaboration check. 7.b.2/1

For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called. 8

Discussion: Note that whether or not the renaming is itself primitive has nothing to do with the renamed subprogram. 8.a

Note that the above rule is only for tagged types. 8.b

8.c Consider the following example:

8.d
```
package P is
    type T is tagged null record;
    function Predefined_Equal(X, Y : T) return Boolean renames "=";
private
    function "="(X, Y : T) return Boolean; -- Override predefined "=".
end P;
```

8.e
```
with P; use P;
package Q is
    function User_Defined_Equal(X, Y : T) return Boolean renames P."=";
end Q;
```

8.f A call on Predefined_Equal will execute the predefined equality operator of T, whereas a call on User_Defined_Equal will execute the body of the overriding declaration in the private part of P.

8.g Thus a renaming allows one to squirrel away a copy of an inherited or predefined subprogram before later overriding it.{*squirrel away*}

Bounded (Run-Time) Errors

8.1/1 {*8652/0027*} {*AI95-00135-01*} {*Program_Error (raised by failure of run-time check)*} {*Storage_Error (raised by failure of run-time check)*} If a subprogram directly or indirectly renames itself, then it is a bounded error to call that subprogram. Possible consequences are that Program_Error or Storage_Error is raised, or that the call results in infinite recursion.

8.g.1/1 **Reason:** {*8652/0027*} {*AI95-00135-01*} This has to be a bounded error, as it is possible for a renaming-as-body appearing in a package body to cause this problem. Thus it is not possible in general to detect this problem at compile time.

NOTES

9 12 A procedure can only be renamed as a procedure. A function whose defining_designator is either an identifier or an operator_symbol can be renamed with either an identifier or an operator_symbol; for renaming as an operator, the subprogram specification given in the renaming_declaration is subject to the rules given in 6.6 for operator declarations. Enumeration literals can be renamed as functions; similarly, attribute_references that denote functions (such as references to Succ and Pred) can be renamed as functions. An entry can only be renamed as a procedure; the new name is only allowed to appear in contexts that allow a procedure name. An entry of a family can be renamed, but an entry family cannot be renamed as a whole.

10 13 The operators of the root numeric types cannot be renamed because the types in the profile are anonymous, so the corresponding specifications cannot be written; the same holds for certain attributes, such as Pos.

11 14 Calls with the new name of a renamed entry are procedure_call_statements and are not allowed at places where the syntax requires an entry_call_statement in conditional_ and timed_entry_calls, nor in an asynchronous_select; similarly, the Count attribute is not available for the new name.

12 15 The primitiveness of a renaming-as-declaration is determined by its profile, and by where it occurs, as for any declaration of (a view of) a subprogram; primitiveness is not determined by the renamed view. In order to perform a dispatching call, the subprogram name has to denote a primitive subprogram, not a non-primitive renaming of a primitive subprogram.

12.a **Reason:** A subprogram_renaming_declaration could more properly be called renaming_as_subprogram_-declaration, since you're renaming something as a subprogram, but you're not necessarily renaming a subprogram. But that's too much of a mouthful. Or, alternatively, we could call it a callable_entity_renaming_declaration, but that's even worse. Not only is it a mouthful, it emphasizes the entity being renamed, rather than the new view, which we think is a bad idea. We'll live with the oddity.

Examples

13 *Examples of subprogram renaming declarations:*

14
```
procedure My_Write(C : in Character) renames Pool(K).Write; -- see 4.1.3
```

15
```
function Real_Plus(Left, Right : Real   ) return Real    renames "+";
function Int_Plus (Left, Right : Integer) return Integer renames "+";
```

16
```
function Rouge return Color renames Red;  -- see 3.5.1
function Rot   return Color renames Red;
function Rosso return Color renames Rouge;
```

17
```
function Next(X : Color) return Color renames Color'Succ; -- see 3.5.1
```

18 *Example of a subprogram renaming declaration with new parameter names:*

19
```
function "*" (X,Y : Vector) return Real renames Dot_Product; -- see 6.1
```

20 *Example of a subprogram renaming declaration with a new default expression:*

21
```
function Minimum(L : Link := Head) return Cell renames Min_Cell; -- see 6.1
```

Extensions to Ada 95

21.a/2 {*8652/0028*} {*AI95-00145-01*} {*extensions to Ada 95*} **Corrigendum:** Allowed a renaming-as-body to be just mode conformant with the specification if the subprogram is not yet frozen.

21.b/2 {*AI95-00218-03*} Overriding_indicator (see 8.3.1) is optionally added to subprogram renamings.

Wording Changes from Ada 95

21.c/2 {*8652/0014*} {*AI95-00064-01*} **Corrigendum:** Described the semantics of renaming-as-body, so that the location of elaboration checks is clear.

21.d/2 {*8652/0027*} {*AI95-00135-01*} **Corrigendum:** Clarified that circular renaming-as-body is illegal (if it can be detected in time) or a bounded error.

21.e/2 {*AI95-00228-01*} **Amendment Correction:** Clarified that renaming a shall-be-overridden subprogram is illegal, as well as renaming-as-body an abstract subprogram.

21.f/2 {*AI95-00423-01*} Added matching rules for null_exclusions.

8.5.5 Generic Renaming Declarations

1 [A generic_renaming_declaration is used to rename a generic unit.]

Syntax

2 generic_renaming_declaration ::=
  generic package defining_program_unit_name **renames** *generic_package_*name;
 | **generic procedure** defining_program_unit_name **renames** *generic_procedure_*name;
 | **generic function** defining_program_unit_name **renames** *generic_function_*name;

Legality Rules

3 The renamed entity shall be a generic unit of the corresponding kind.

Static Semantics

4 A generic_renaming_declaration declares a new view [of the renamed generic unit].

NOTES

5 16 Although the properties of the new view are the same as those of the renamed view, the place where the generic_renaming_declaration occurs may affect the legality of subsequent renamings and instantiations that denote the generic_renaming_declaration, in particular if the renamed generic unit is a library unit (see 10.1.1).

Examples

6 *Example of renaming a generic unit:*

7
```
generic package Enum_IO renames Ada.Text_IO.Enumeration_IO;  -- see A.10.10
```

Extensions to Ada 83

7.a {*extensions to Ada 83*} Renaming of generic units is new to Ada 95. It is particularly important for renaming child library units that are generic units. For example, it might be used to rename Numerics.Generic_Elementary_Functions as simply Generic_Elementary_Functions, to match the name for the corresponding Ada-83-based package.

Wording Changes from Ada 83

7.b The information in RM83-8.6, “The Package Standard,” has been updated for the child unit feature, and moved to Annex A, except for the definition of “predefined type,” which has been moved to 3.2.1.

8.6 The Context of Overload Resolution

1 [{*overload resolution*} Because declarations can be overloaded, it is possible for an occurrence of a usage name to have more than one possible interpretation; in most cases, ambiguity is disallowed. This clause describes how the possible interpretations resolve to the actual interpretation.

2 {*overloading rules*} Certain rules of the language (the Name Resolution Rules) are considered “overloading rules”. If a possible interpretation violates an overloading rule, it is assumed not to be the intended interpretation; some other possible interpretation is assumed to be the actual interpretation. On the other hand, violations of non-overloading rules do not affect which interpretation is chosen; instead, they cause the construct to be illegal. To be legal, there usually has to be exactly one acceptable interpretation of a construct that is a “complete context”, not counting any nested complete contexts.

3 {*grammar (resolution of ambiguity)*} The syntax rules of the language and the visibility rules given in 8.3 determine the possible interpretations. Most type checking rules (rules that require a particular type, or a particular class of types, for example) are overloading rules. Various rules for the matching of formal and actual parameters are overloading rules.]

Language Design Principles

3.a The type resolution rules are intended to minimize the need for implicit declarations and preference rules associated with implicit conversion and dispatching operations.

Name Resolution Rules

4 {*complete context*} [Overload resolution is applied separately to each *complete context*, not counting inner complete contexts.] Each of the following constructs is a *complete context*:

5 • A context_item.

6 • A declarative_item or declaration.

6.a **Ramification:** A loop_parameter_specification is a declaration, and hence a complete context.

7 • A statement.

8 • A pragma_argument_association.

8.a **Reason:** We would make it the whole pragma, except that certain pragma arguments are allowed to be ambiguous, and ambiguity applies to a complete context.

9 • The expression of a case_statement.

9.a **Ramification:** This means that the expression is resolved without looking at the choices.

10 {*interpretation (of a complete context)*} {*overall interpretation (of a complete context)*} An (overall) *interpretation* of a complete context embodies its meaning, and includes the following information about the constituents of the complete context, not including constituents of inner complete contexts:

11 • for each constituent of the complete context, to which syntactic categories it belongs, and by which syntax rules; and

11.a **Ramification:** Syntactic categor*ies* is plural here, because there are lots of trivial productions — an expression might also be all of the following, in this order: identifier, name, primary, factor, term, simple_expression, and relation. Basically, we're trying to capture all the information in the parse tree here, without using compiler-writer's jargon like “parse tree”.

12 • for each usage name, which declaration it denotes (and, therefore, which view and which entity it denotes); and

12.a/2 **Ramification:** {*AI95-00382-01*} In most cases, a usage name denotes the view declared by the denoted declaration. However, in certain cases, a usage name that denotes a declaration and appears inside the declarative region of that same declaration, denotes the current instance of the declaration. For example, within a task_body other than in an access_definition, a usage name that denotes the task_type_declaration denotes the object containing the currently executing task, and not the task type declared by the declaration.

- for a complete context that is a declarative_item, whether or not it is a completion of a declaration, and (if so) which declaration it completes. 13

Ramification: Unfortunately, we are not confident that the above list is complete. We'll have to live with that. 13.a

To be honest: For "possible" interpretations, the above information is tentative. 13.b

Discussion: A possible interpretation (an *input* to overload resolution) contains information about what a usage name *might* denote, but what it actually *does* denote requires overload resolution to determine. Hence the term "tentative" is needed for possible interpretations; otherwise, the definition would be circular. 13.c

{*possible interpretation*} A *possible interpretation* is one that obeys the syntax rules and the visibility rules. {*acceptable interpretation*} {*resolve (overload resolution)*} {*interpretation (overload resolution)*} An *acceptable interpretation* is a possible interpretation that obeys the *overloading rules*[, that is, those rules that specify an expected type or expected profile, or specify how a construct shall *resolve* or be *interpreted*.] 14

To be honest: One rule that falls into this category, but does not use the above-mentioned magic words, is the rule about numbers of parameter associations in a call (see 6.4). 14.a

Ramification: The Name Resolution Rules are the ones that appear under the Name Resolution Rules heading. Some Syntax Rules are written in English, instead of BNF. No rule is a Syntax Rule or Name Resolution Rule unless it appears under the appropriate heading. 14.b

{*interpretation (of a constituent of a complete context)*} The *interpretation* of a constituent of a complete context is determined from the overall interpretation of the complete context as a whole. [Thus, for example, "interpreted as a function_call," means that the construct's interpretation says that it belongs to the syntactic category function_call.] 15

{*denote*} [Each occurrence of] a usage name *denotes* the declaration determined by its interpretation. It also denotes the view declared by its denoted declaration, except in the following cases: 16

Ramification: As explained below, a pragma argument is allowed to be ambiguous, so it can denote several declarations, and all of the views declared by those declarations. 16.a

- {*AI95-00382-01*} {*current instance (of a type)*} If a usage name appears within the declarative region of a type_declaration and denotes that same type_declaration, then it denotes the *current instance* of the type (rather than the type itself); the current instance of a type is the object or value of the type that is associated with the execution that evaluates the usage name. This rule does not apply if the usage name appears within the subtype_mark of an access_definition for an access-to-object type, or within the subtype of a parameter or result of an access-to-subprogram type. 17/2

Reason: {*AI95-00382-01*} This is needed, for example, for references to the Access attribute from within the type_declaration. Also, within a task_body or protected_body, we need to be able to denote the current task or protected object. (For a single_task_declaration or single_protected_declaration, the rule about current instances is not needed.) We exclude anonymous access types so that they can be used to create self-referencing types in the natural manner (otherwise such types would be illegal). 17.a/2

Discussion: {*AI95-00382-01*} The phrase "within the subtype_mark" in the "this rule does not apply" part is intended to cover a case like **access** T'Class appearing within the declarative region of T: here T denotes the type, not the current instance. 17.b/2

- {*current instance (of a generic unit)*} If a usage name appears within the declarative region of a generic_declaration (but not within its generic_formal_part) and it denotes that same generic_declaration, then it denotes the *current instance* of the generic unit (rather than the generic unit itself). See also 12.3. 18

To be honest: The current instance of a generic unit is the instance created by whichever generic_instantiation is of interest at any given time. 18.a

Ramification: Within a generic_formal_part, a name that denotes the generic_declaration denotes the generic unit, which implies that it is not overloadable. 18.b

A usage name that denotes a view also denotes the entity of that view. 19

19.a **Ramification:** Usually, a usage name denotes only one declaration, and therefore one view and one entity.

20/2 {*AI95-00231-01*} {*expected type* [distributed]} The *expected type* for a given expression, name, or other construct determines, according to the *type resolution rules* given below, the types considered for the construct during overload resolution. {*type resolution rules*} [The type resolution rules provide support for class-wide programming, universal literals, dispatching operations, and anonymous access types:]

20.a **Ramification:** Expected types are defined throughout the RM95. The most important definition is that, for a subprogram, the expected type for the actual parameter is the type of the formal parameter.

20.b The type resolution rules are trivial unless either the actual or expected type is universal, class-wide, or of an anonymous access type.

21 • {*type resolution rules (if any type in a specified class of types is expected)* [partial]} {*type resolution rules (if expected type is universal or class-wide)* [partial]} If a construct is expected to be of any type in a class of types, or of the universal or class-wide type for a class, then the type of the construct shall resolve to a type in that class or to a universal type that covers the class.

21.a **Ramification:** This matching rule handles (among other things) cases like the Val attribute, which denotes a function that takes a parameter of type *universal_integer*.

21.b/1 The last part of the rule, "or to a universal type that covers the class" implies that if the expected type for an expression is *universal_fixed*, then an expression whose type is *universal_real* (such as a real literal) is OK.

22 • {*type resolution rules (if expected type is specific)* [partial]} If the expected type for a construct is a specific type *T*, then the type of the construct shall resolve either to *T*, or:

22.a **Ramification:** {*Beaujolais effect* [partial]} This rule is *not* intended to create a preference for the specific type — such a preference would cause Beaujolais effects.

23 • to *T*'Class; or

23.a **Ramification:** This will only be legal as part of a call on a dispatching operation; see 3.9.2, "Dispatching Operations of Tagged Types". Note that that rule is not a Name Resolution Rule.

24 • to a universal type that covers *T*; or

25/2 • {*AI95-00230-01*} {*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00409-01*} when *T* is a specific anonymous access-to-object type (see 3.10) with designated type *D*, to an access-to-object type whose designated type is *D*'Class or is covered by *D*; or

25.a/2 *This paragraph was deleted.*{*AI95-00409-01*}

25.b **Ramification:** The case where the actual is access-to-*D*'Class will only be legal as part of a call on a dispatching operation; see 3.9.2, "Dispatching Operations of Tagged Types". Note that that rule is not a Name Resolution Rule.

25.1/2 • {*AI95-00254-01*} {*AI95-00409-01*} when *T* is an anonymous access-to-subprogram type (see 3.10), to an access-to-subprogram type whose designated profile is type-conformant with that of *T*.

26 {*expected profile* [distributed]} In certain contexts, [such as in a subprogram_renaming_declaration,] the Name Resolution Rules define an *expected profile* for a given name; {*profile resolution rule (name with a given expected profile)*} in such cases, the name shall resolve to the name of a callable entity whose profile is type conformant with the expected profile. {*type conformance (required)*}

26.a **Ramification:** The parameter and result *sub*types are not used in overload resolution. Only type conformance of profiles is considered during overload resolution. Legality rules generally require at least mode-conformance in addition, but those rules are not used in overload resolution.

Legality Rules

27/2 {*AI95-00332-01*} {*single (class expected type)*} When a construct is one that requires that its expected type be a *single* type in a given class, the type of the construct shall be determinable solely from the context in which the construct appears, excluding the construct itself, but using the requirement that it be in the given class. Furthermore, the context shall not be one that expects any type in some class that contains types of the given class; in particular, the construct shall not be the operand of a type_conversion.

Ramification: {*AI95-00230-01*} For example, the expected type for a string literal is required to be a single string type. But the expected type for the operand of a type_conversion is any type. Therefore, a string literal is not allowed as the operand of a type_conversion. This is true even if there is only one string type in scope (which is never the case). The reason for these rules is so that the compiler will not have to search “everywhere” to see if there is exactly one type in a class in scope. 27.a/2

Discussion: {*AI95-00332-01*} The first sentence is carefully worded so that it only mentions “expected type” as part of identifying the interesting case, but doesn't require that the context actually provide such an expected type. This allows such constructs to be used inside of constructs that don't provide an expected type (like qualified expressions and renames). Otherwise, such constructs wouldn't allow aggregates, 'Access, and so on. 27.b/2

A complete context shall have at least one acceptable interpretation; if there is exactly one, then that one is chosen. 28

Ramification: This, and the rule below about ambiguity, are the ones that suck in all the Syntax Rules and Name Resolution Rules as compile-time rules. Note that this and the ambiguity rule have to be Legality Rules. 28.a

{*preference (for root numeric operators and ranges)*} There is a *preference* for the primitive operators (and ranges) of the root numeric types *root_integer* and *root_real*. In particular, if two acceptable interpretations of a constituent of a complete context differ only in that one is for a primitive operator (or range) of the type *root_integer* or *root_real*, and the other is not, the interpretation using the primitive operator (or range) of the root numeric type is *preferred*. 29

Reason: The reason for this preference is so that expressions involving literals and named numbers can be unambiguous. For example, without the preference rule, the following would be ambiguous: 29.a

29.b/1
```
N : constant := 123;
if N > 100 then -- Preference for root_integer ">" operator.
    ...
end if;
```

For a complete context, if there is exactly one overall acceptable interpretation where each constituent's interpretation is the same as or preferred (in the above sense) over those in all other overall acceptable interpretations, then that one overall acceptable interpretation is chosen. {*ambiguous*} Otherwise, the complete context is *ambiguous*. 30

A complete context other than a pragma_argument_association shall not be ambiguous. 31

A complete context that is a pragma_argument_association is allowed to be ambiguous (unless otherwise specified for the particular pragma), but only if every acceptable interpretation of the pragma argument is as a name that statically denotes a callable entity. {*denote (name used as a pragma argument)* [partial]} Such a name denotes all of the declarations determined by its interpretations, and all of the views declared by these declarations. 32

Ramification: {*AI95-00224-01*} This applies to Inline, Suppress, Import, Export, and Convention pragmas. For example, it is OK to say “**pragma** Export(C, Entity_Name => P.Q);”, even if there are two directly visible P's, and there are two Q's declared in the visible part of each P. In this case, P.Q denotes four different declarations. This rule also applies to certain pragmas defined in the Specialized Needs Annexes. It almost applies to Pure, Elaborate_Body, and Elaborate_All pragmas, but those can't have overloading for other reasons. 32.a/2

Note that if a pragma argument denotes a *call* to a callable entity, rather than the entity itself, this exception does not apply, and ambiguity is disallowed. 32.b

Note that we need to carefully define which pragma-related rules are Name Resolution Rules, so that, for example, a pragma Inline does not pick up subprograms declared in enclosing declarative regions, and therefore make itself illegal. 32.c

We say “statically denotes” in the above rule in order to avoid having to worry about how many times the name is evaluated, in case it denotes more than one callable entity. 32.d

NOTES

17 If a usage name has only one acceptable interpretation, then it denotes the corresponding entity. However, this does not mean that the usage name is necessarily legal since other requirements exist which are not considered for overload resolution; for example, the fact that an expression is static, whether an object is constant, mode and subtype conformance rules, freezing rules, order of elaboration, and so on. 33

34 Similarly, subtypes are not considered for overload resolution (the violation of a constraint does not make a program illegal but raises an exception during program execution).

Incompatibilities With Ada 83

34.a {*incompatibilities with Ada 83*} {*Beaujolais effect* [partial]} The new preference rule for operators of root numeric types is upward incompatible, but only in cases that involved *Beaujolais* effects in Ada 83. Such cases are ambiguous in Ada 95.

Extensions to Ada 83

34.b {*extensions to Ada 83*} The rule that allows an expected type to match an actual expression of a universal type, in combination with the new preference rule for operators of root numeric types, subsumes the Ada 83 "implicit conversion" rules for universal types.

Wording Changes from Ada 83

34.c In Ada 83, it is not clear what the “syntax rules” are. AI83-00157 states that a certain textual rule is a syntax rule, but it's still not clear how one tells in general which textual rules are syntax rules. We have solved the problem by stating exactly which rules are syntax rules — the ones that appear under the “Syntax” heading.

34.d RM83 has a long list of the “forms” of rules that are to be used in overload resolution (in addition to the syntax rules). It is not clear exactly which rules fall under each form. We have solved the problem by explicitly marking all rules that are used in overload resolution. Thus, the list of kinds of rules is unnecessary. It is replaced with some introductory (intentionally vague) text explaining the basic idea of what sorts of rules are overloading rules.

34.e It is not clear from RM83 what information is embodied in a “meaning” or an “interpretation.” “Meaning” and “interpretation” were intended to be synonymous; we now use the latter only in defining the rules about overload resolution. “Meaning” is used only informally. This clause attempts to clarify what is meant by “interpretation.”

34.f For example, RM83 does not make it clear that overload resolution is required in order to match subprogram_bodies with their corresponding declarations (and even to tell whether a given subprogram_body is the completion of a previous declaration). Clearly, the information needed to do this is part of the “interpretation” of a subprogram_body. The resolution of such things is defined in terms of the “expected profile” concept. Ada 95 has some new cases where expected profiles are needed — the resolution of P'Access, where P might denote a subprogram, is an example.

34.g RM83-8.7(2) might seem to imply that an interpretation embodies information about what is denoted by each usage name, but not information about which syntactic category each construct belongs to. However, it seems necessary to include such information, since the Ada grammar is highly ambiguous. For example, X(Y) might be a function_call or an indexed_component, and no context-free/syntactic information can tell the difference. It seems like we should view X(Y) as being, for example, “interpreted as a function_call” (if that's what overload resolution decides it is). Note that there are examples where the denotation of each usage name does not imply the syntactic category. However, even if that were not true, it seems that intuitively, the interpretation includes that information. Here's an example:

34.h
```
type T;
type A is access T;
type T is array(Integer range 1..10) of A;
I : Integer := 3;
function F(X : Integer := 7) return A;
Y : A := F(I); -- Ambiguous? (We hope so.)
```

34.i/1 Consider the declaration of Y (a complete context). In the above example, overload resolution can easily determine the declaration, and therefore the entity, denoted by Y, A, F, and I. However, given all of that information, we still don't know whether F(I) is a function_call or an indexed_component whose prefix is a function_call. (In the latter case, it is equivalent to F(7).**all**(I).)

34.j It seems clear that the declaration of Y ought to be considered ambiguous. We describe that by saying that there are two interpretations, one as a function_call, and one as an indexed_component. These interpretations are both acceptable to the overloading rules. Therefore, the complete context is ambiguous, and therefore illegal.

34.k {*Beaujolais effect* [partial]} It is the intent that the Ada 95 preference rule for root numeric operators is more locally enforceable than that of RM83-4.6(15). It should also eliminate interpretation shifts due to the addition or removal of a use_clause (the so called *Beaujolais* effect).

34.l/2 {*AI95-00114-01*} RM83-8.7 seems to be missing some complete contexts, such as pragma_argument_associations, declarative_items that are not declarations or aspect_clauses, and context_items. We have added these, and also replaced the “must be determinable” wording of RM83-5.4(3) with the notion that the expression of a case_statement is a complete context.

34.m Cases like the Val attribute are now handled using the normal type resolution rules, instead of having special cases that explicitly allow things like “any integer type.”

Incompatibilities With Ada 95

{*AI95-00409-01*} {*incompatibilities with Ada 95*} Ada 95 allowed name resolution to distinguish between anonymous access-to-variable and access-to-constant types. This is similar to distinguishing between subprograms with **in** and **in out** parameters, which is known to be bad. Thus, that part of the rule was dropped as we now have anonymous access-to-constant types, making this much more likely. 34.n/2

34.o/2
```
type Cacc is access constant Integer;
procedure Proc (Acc : access Integer) ...
procedure Proc (Acc : Cacc) ...
List : Cacc := ...;
Proc (List); -- OK in Ada 95, ambiguous in Ada 2005.
```

If there is any code like this (such code should be rare), it will be ambiguous in Ada 2005. 34.p/2

Extensions to Ada 95

{*AI95-00230-01*} {*AI95-00231-01*} {*AI95-00254-01*} {*extensions to Ada 95*} Generalized the anonymous access resolution rules to support the new capabilities of anonymous access types (that is, access-to-subprogram and access-to-constant). 34.q/2

{*AI95-00382-01*} We now allow the creation of self-referencing types via anonymous access types. This is an extension in unusual cases involving task and protected types. For example: 34.r/2

34.s/2
```
task type T;
```

34.t/2
```
task body T is
   procedure P (X : access T) is -- Illegal in Ada 95, legal in Ada 2005
      ...
   end P;
begin
   ...
end T;
```

Wording Changes from Ada 95

{*AI95-00332-01*} Corrected the “single expected type” so that it works in contexts that don't have expected types (like object renames and qualified expressions). This fixes a hole in Ada 95 that appears to prohibit using aggregates, 'Access, character literals, string literals, and allocators in qualified expressions. 34.u/2

Section 9: Tasks and Synchronization

1 {*execution (Ada program)* [partial]} The execution of an Ada program consists of the execution of one or more *tasks*. {*task*} {*interaction (between tasks)*} Each task represents a separate thread of control that proceeds independently and concurrently between the points where it *interacts* with other tasks. The various forms of task interaction are described in this section, and include: {*parallel processing: See task*} {*synchronization*} {*concurrent processing: See task*} {*intertask communication: See also task*}

1.a **To be honest:** The execution of an Ada program consists of the execution of one or more partitions (see 10.2), each of which in turn consists of the execution of an environment task and zero or more subtasks.

- 2 the activation and termination of a task;
- 3 {*protected object*} a call on a protected subprogram of a *protected object*, providing exclusive read-write access, or concurrent read-only access to shared data;
- 4 a call on an entry, either of another task, allowing for synchronous communication with that task, or of a protected object, allowing for asynchronous communication with one or more other tasks using that same protected object;
- 5 a timed operation, including a simple delay statement, a timed entry call or accept, or a timed asynchronous select statement (see next item);
- 6 an asynchronous transfer of control as part of an asynchronous select statement, where a task stops what it is doing and begins execution at a different point in response to the completion of an entry call or the expiration of a delay;
- 7 an abort statement, allowing one task to cause the termination of another task.

8 In addition, tasks can communicate indirectly by reading and updating (unprotected) shared variables, presuming the access is properly synchronized through some other kind of task interaction.

Static Semantics

9 {*task unit*} The properties of a task are defined by a corresponding task declaration and task_body, which together define a program unit called a *task unit*.

Dynamic Semantics

10 Over time, tasks proceed through various *states*. {*task state (inactive)* [partial]} {*inactive (a task state)*} {*task state (blocked)* [partial]} {*blocked (a task state)*} {*task state (ready)* [partial]} {*ready (a task state)*} {*task state (terminated)* [partial]} {*terminated (a task state)*} A task is initially *inactive*; upon activation, and prior to its *termination* it is either *blocked* (as part of some task interaction) or *ready* to run. {*execution resource (required for a task to run)*} While ready, a task competes for the available *execution resources* that it requires to run.

10.a **Discussion:** {*task dispatching policy*} {*dispatching policy for tasks*} The means for selecting which of the ready tasks to run, given the currently available execution resources, is determined by the *task dispatching policy* in effect, which is generally implementation defined, but may be controlled by pragmas and operations defined in the Real-Time Annex (see D.2 and D.5).

NOTES

11 1 Concurrent task execution may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a given task are performed by different physical processors acting in parallel, it may choose to perform them in this way.

Wording Changes from Ada 83

11.a The introduction has been rewritten.

11.b We use the term "concurrent" rather than "parallel" when talking about logically independent execution of threads of control. The term "parallel" is reserved for referring to the situation where multiple physical processors run simultaneously.

9.1 Task Units and Task Objects

1 {*task declaration*} A task unit is declared by a *task declaration*, which has a corresponding task_body. A task declaration may be a task_type_declaration, in which case it declares a named task type; alternatively, it may be a single_task_declaration, in which case it defines an anonymous task type, as well as declaring a named task object of that type.

Syntax

2/2 {*AI95-00345-01*} task_type_declaration ::=
  task type defining_identifier [known_discriminant_part] [**is**
   [**new** interface_list **with**]
   task_definition];

3/2 {*AI95-00399-01*} single_task_declaration ::=
  task defining_identifier [**is**
   [**new** interface_list **with**]
   task_definition];

4 task_definition ::=
   {task_item}
  [**private**
   {task_item}]
  end [*task*_identifier]

5/1 {*8652/0009*} {*AI95-00137-01*} task_item ::= entry_declaration | aspect_clause

6 task_body ::=
  task body defining_identifier **is**
   declarative_part
  begin
   handled_sequence_of_statements
  end [*task*_identifier];

7 If a *task*_identifier appears at the end of a task_definition or task_body, it shall repeat the defining_identifier.

Legality Rules

8/2 *This paragraph was deleted.*{*AI95-00345-01*}

8.a/2 *This paragraph was deleted.*

Static Semantics

9 A task_definition defines a task type and its first subtype. {*visible part (of a task unit)* [partial]} The first list of task_items of a task_definition, together with the known_discriminant_part, if any, is called the visible part of the task unit. [{*private part (of a task unit)* [partial]} The optional list of task_items after the reserved word **private** is called the private part of the task unit.]

9.a **Proof:** Private part is defined in Section 8.

9.1/1 {*8652/0029*} {*AI95-00116-01*} For a task declaration without a task_definition, a task_definition without task_items is assumed.

{*AI95-00345-01*} {*AI95-00397-01*} {*AI95-00399-01*} {*AI95-00419-01*} For a task declaration with an interface_list, the task type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the task type or an access parameter designating the task type, and there is an entry_declaration for a single entry with the same identifier within the task declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming task entry.{*implemented (by a task entry)* [partial]} {*type conformance (required)*} 9.2/2

Ramification: The inherited subprograms can only come from an interface given as part of the task declaration. 9.b/2

Legality Rules

{*AI95-00345-01*} {*requires a completion (task_declaration})* [partial]} A task declaration requires a completion[, which shall be a task_body,] and every task_body shall be the completion of some task declaration. 9.3/2

To be honest: The completion can be a pragma Import, if the implementation supports it. 9.c/2

{*AI95-00345-01*} {*AI95-00399-01*} [Each *interface_*subtype_mark of an interface_list appearing within a task declaration shall denote a limited interface type that is not a protected interface.] 9.4/2

Proof: 3.9.4 requires that an interface_list only name interface types, and limits the descendants of the various kinds of interface types. Only a limited, task, or synchronized interface can have a task type descendant. Nonlimited or protected interfaces are not allowed, as they offer operations that a task does not have. 9.d/2

{*AI95-00397-01*} The prefixed view profile of an explicitly declared primitive subprogram of a tagged task type shall not be type conformant with any entry of the task type, if the first parameter of the subprogram is of the task type or is an access parameter designating the task type. 9.5/2

Reason: This prevents the existence of two operations with the same name and profile which could be called with a prefixed view. If the operation was inherited, this would be illegal by the following rules; this rule puts inherited and non-inherited routines on the same footing. Note that this only applies to tagged task types (that is, those with an interface in their declaration); we do that as there is no problem with prefixed view calls of primitive operations for “normal” task types, and having this rule apply to all tasks would be incompatible with Ada 95. 9.e/2

{*AI95-00345-01*} {*AI95-00399-01*} For each primitive subprogram inherited by the type declared by a task declaration, at most one of the following shall apply: 9.6/2

- {*AI95-00345-01*} the inherited subprogram is overridden with a primitive subprogram of the task type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or{*subtype conformance (required)*} 9.7/2
- {*AI95-00345-01*} {*AI95-00397-01*} the inherited subprogram is implemented by a single entry of the task type; in which case its prefixed view profile shall be subtype conformant with that of the task entry. {*subtype conformance (required)*} 9.8/2

Ramification: An entry may implement two subprograms from the ancestors, one whose first parameter is of type *T* and one whose first parameter is of type **access** *T*. That doesn't cause implementation problems because “implemented by” (unlike “overridden’) probably entails the creation of wrappers. 9.f/2

If neither applies, the inherited subprogram shall be a null procedure. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 9.9/2

Reason: Each inherited subprogram can only have a single implementation (either from overriding a subprogram or implementing an entry), and must have an implementation unless the subprogram is a null procedure. 9.g/2

Dynamic Semantics

10 [{*elaboration (task declaration)* [partial]}] The elaboration of a task declaration elaborates the task_definition. {*elaboration (single_task_declaration)* [partial]} The elaboration of a single_task_declaration also creates an object of an (anonymous) task type.]

10.a **Proof:** This is redundant with the general rules for the elaboration of a full_type_declaration and an object_declaration.

11 {*elaboration (task_definition)* [partial]} [The elaboration of a task_definition creates the task type and its first subtype;] it also includes the elaboration of the entry_declarations in the given order.

12/1 {*8652/0009*} {*AI95-00137-01*} {*initialization (of a task object)* [partial]} As part of the initialization of a task object, any aspect_clauses and any per-object constraints associated with entry_declarations of the corresponding task_definition are elaborated in the given order.

12.a/1 **Reason:** The only aspect_clauses defined for task entries are ones that specify the Address of an entry, as part of defining an interrupt entry. These clearly need to be elaborated per-object, not per-type. Normally the address will be a function of a discriminant, if such an Address clause is in a task type rather than a single task declaration, though it could rely on a parameterless function that allocates sequential interrupt vectors.

12.b We do not mention representation pragmas, since each pragma may have its own elaboration rules.

13 {*elaboration (task_body)* [partial]} The elaboration of a task_body has no effect other than to establish that tasks of the type can from then on be activated without failing the Elaboration_Check.

14 [The execution of a task_body is invoked by the activation of a task of the corresponding type (see 9.2).]

15 The content of a task object of a given task type includes:

16 • The values of the discriminants of the task object, if any;

17 • An entry queue for each entry of the task object;

17.a **Ramification:** "For each entry" implies one queue for each single entry, plus one for each entry of each entry family.

18 • A representation of the state of the associated task.

NOTES

19/2 2 {*AI95-00382-01*} Other than in an access_definition, the name of a task unit within the declaration or body of the task unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding task type (and thus the name cannot be used as a subtype_mark).

19.a/2 **Discussion:** {*AI95-00382-01*} It can be used as a subtype_mark in an anonymous access type. In addition, it is possible to refer to some other subtype of the task type within its body, presuming such a subtype has been declared between the task_type_declaration and the task_body.

20 3 The notation of a selected_component can be used to denote a discriminant of a task (see 4.1.3). Within a task unit, the name of a discriminant of the task type denotes the corresponding discriminant of the current instance of the unit.

21/2 4 {*AI95-00287-01*} A task type is a limited type (see 7.5), and hence precludes use of assignment_statements and predefined equality operators. If an application needs to store and exchange task identities, it can do so by defining an access type designating the corresponding task objects and by using access values for identification purposes. Assignment is available for such an access type as for any access type. Alternatively, if the implementation supports the Systems Programming Annex, the Identity attribute can be used for task identification (see C.7.1).

Examples

22 *Examples of declarations of task types:*

23
```
task type Server is
   entry Next_Work_Item(WI : in Work_Item);
   entry Shut_Down;
end Server;
```

24/2
```
{AI95-00433-01} task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is
      new Serial_Device with  -- see 3.9.4
   entry Read (C : out Character);
   entry Write(C : in  Character);
end Keyboard_Driver;
```

Examples of declarations of single tasks: 25

26
```
task Controller is
   entry Request(Level)(D : Item);  -- a family of entries
end Controller;
```

27
```
task Parser is
   entry Next_Lexeme(L : in  Lexical_Element);
   entry Next_Action(A : out Parser_Action);
end;
```

28
```
task User;  -- has no entries
```

Examples of task objects: 29

30
```
Agent    : Server;
Teletype : Keyboard_Driver(TTY_ID);
Pool     : array(1 .. 10) of Keyboard_Driver;
```

Example of access type designating task objects: 31

32
```
type Keyboard is access Keyboard_Driver;
Terminal : Keyboard := new Keyboard_Driver(Term_ID);
```

Extensions to Ada 83

32.a/1 {*extensions to Ada 83*} The syntax rules for task declarations are modified to allow a known_discriminant_part, and to allow a private part. They are also modified to allow entry_declarations and aspect_clauses to be mixed.

Wording Changes from Ada 83

32.b The syntax rules for tasks have been split up according to task types and single tasks. In particular: The syntax rules for task_declaration and task_specification are removed. The syntax rules for task_type_declaration, single_task_declaration, task_definition and task_item are new.

32.c The syntax rule for task_body now uses the nonterminal handled_sequence_of_statements.

32.d The declarative_part of a task_body is now required; that doesn't make any real difference, because a declarative_part can be empty.

Extensions to Ada 95

32.e/2 {*AI95-00345-01*} {*AI95-00397-01*} {*AI95-00399-01*} {*AI95-00419-01*} {*extensions to Ada 95*} Task types and single tasks can be derived from one or more interfaces. Entries of the task type can implement the primitive operations of an interface. Overriding_indicators can be used to specify whether or not an entry implements a primitive operation.

Wording Changes from Ada 95

32.f/2 {*8652/0029*} {*AI95-00116-01*} **Corrigendum:** Clarified that a task type has an implicit empty task_definition if none is given.

32.g/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Changed representation clauses to aspect clauses to reflect that they are used for more than just representation.

32.h/2 {*AI95-00287-01*} Revised the note on operations of task types to reflect that limited types do have an assignment operation, but not copying (assignment_statements).

32.i/2 {*AI95-00382-01*} Revised the note on use of the name of a task type within itself to reflect the exception for anonymous access types.

9.2 Task Execution - Task Activation

Dynamic Semantics

1 {*execution (task)* [partial]} The execution of a task of a given task type consists of the execution of the corresponding task_body. {*execution (task_body)* [partial]} {*task (execution)*} {*activation (of a task)*} {*task (activation)*} The initial part of this execution is called the *activation* of the task; it consists of the elaboration of the declarative_part of the task_body. {*activation failure*} Should an exception be

propagated by the elaboration of its declarative_part, the activation of the task is defined to have *failed*, and it becomes a completed task.

2/2 {*AI95-00416-01*} A task object (which represents one task) can be a part of a stand-alone object, of an object created by an allocator, or of an anonymous object of a limited type, or a coextension of one of these. All tasks that are part or coextensions of any of the stand-alone objects created by the elaboration of object_declarations (or generic_associations of formal objects of mode **in**) of a single declarative region are activated together. All tasks that are part or coextensions of a single object that is not a stand-alone object are activated together.

2.a **Discussion:** The initialization of an object_declaration or allocator can indirectly include the creation of other objects that contain tasks. For example, the default expression for a subcomponent of an object created by an allocator might call a function that evaluates a completely different allocator. Tasks created by the two allocators are *not* activated together.

3/2 {*AI95-00416-01*} For the tasks of a given declarative region, the activations are initiated within the context of the handled_sequence_of_statements (and its associated exception_handlers if any — see 11.2), just prior to executing the statements of the handled_sequence_of_statements. [For a package without an explicit body or an explicit handled_sequence_of_statements, an implicit body or an implicit null_statement is assumed, as defined in 7.2.]

3.a **Ramification:** If Tasking_Error is raised, it can be handled by handlers of the handled_sequence_of_statements.

4/2 {*AI95-00416-01*} For tasks that are part or coextensions of a single object that is not a stand-alone object, activations are initiated after completing any initialization of the outermost object enclosing these tasks, prior to performing any other operation on the outermost object. In particular, for tasks that are part or coextensions of the object created by the evaluation of an allocator, the activations are initiated as the last step of evaluating the allocator, prior to returning the new access value. For tasks that are part or coextensions of an object that is the result of a function call, the activations are not initiated until after the function returns.

4.a/2 **Discussion:** {*AI95-00416-01*} The intent is that "temporary" objects with task parts (or coextensions) are treated similarly to an object created by an allocator. The "whole" object is initialized, and then all of the task parts (including the coextensions) are activated together. Each such "whole" object has its own task activation sequence, involving the activating task being suspended until all the new tasks complete their activation.

5 {*activator (of a task)*} {*blocked (waiting for activations to complete)* [partial]} The task that created the new tasks and initiated their activations (the *activator*) is blocked until all of these activations complete (successfully or not). {*Tasking_Error (raised by failure of run-time check)*} Once all of these activations are complete, if the activation of any of the tasks has failed [(due to the propagation of an exception)], Tasking_Error is raised in the activator, at the place at which it initiated the activations. Otherwise, the activator proceeds with its execution normally. Any tasks that are aborted prior to completing their activation are ignored when determining whether to raise Tasking_Error.

5.a **Ramification:** Note that a task created by an allocator does not necessarily depend on its activator; in such a case the activator's termination can precede the termination of the newly created task.

5.b **Discussion:** Tasking_Error is raised only once, even if two or more of the tasks being activated fail their activation.

5.c/2 **To be honest:** {*AI95-00265-01*} The pragma Partition_Elaboration_Policy (see H.6) can be used to defer task activation to a later point, thus changing many of these rules.

6 Should the task that created the new tasks never reach the point where it would initiate the activations (due to an abort or the raising of an exception), the newly created tasks become terminated and are never activated.

NOTES

7 5 An entry of a task can be called before the task has been activated.

8 6 If several tasks are activated together, the execution of any of these tasks need not await the end of the activation of the other tasks.

7 A task can become completed during its activation either because of an exception or because it is aborted (see 9.8). 9

Examples

Example of task activation: 10

```
procedure P is                                                     11
   A, B : Server;     -- elaborate the task objects A, B
   C    : Server;     -- elaborate the task object C
begin
   -- the tasks A, B, C are activated together before the first statement
   ...
end;
```

Wording Changes from Ada 83

We have replaced the term *suspended* with *blocked*, since we didn't want to consider a task blocked when it was simply competing for execution resources. "Suspended" is sometimes used more generally to refer to tasks that are not actually running on some processor, due to the lack of resources. 11.a

This clause has been rewritten in an attempt to improve presentation. 11.b

Wording Changes from Ada 95

{*AI95-00416-01*} Adjusted the wording for activating tasks to handle the case of anonymous function return objects. This is critical; we don't want to be waiting for the tasks in a return object when we exit the function normally. 11.c/2

9.3 Task Dependence - Termination of Tasks

Dynamic Semantics

{*dependence (of a task on a master)*} {*task (dependence)*} {*task (completion)*} {*task (termination)*} Each task (other than an environment task — see 10.2) *depends* on one or more masters (see 7.6.1), as follows: 1

- If the task is created by the evaluation of an allocator for a given access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type. 2
- If the task is created by the elaboration of an object_declaration, it depends on each master that includes this elaboration. 3
- {*AI95-00416-01*} Otherwise, the task depends on the master of the outermost object of which it is a part (as determined by the accessibility level of that object — see 3.10.2 and 7.6.1), as well as on any master whose execution includes that of the master of the outermost object. 3.1/2

Ramification: {*AI95-00416-01*} The master of a task created by a return statement changes when the accessibility of the return object changes. Note that its activation happens, if at all, only after the function returns and all accessibility level changes have occurred. 3.a/2

{*dependence (of a task on another task)*} Furthermore, if a task depends on a given master, it is defined to depend on the task that executes the master, and (recursively) on any master of that task. 4

Discussion: Don't confuse these kinds of dependences with the dependences among compilation units defined in 10.1.1, “Compilation Units - Library Units”. 4.a

A task is said to be *completed* when the execution of its corresponding task_body is completed. A task is said to be *terminated* when any finalization of the task_body has been performed (see 7.6.1). [The first step of finalizing a master (including a task_body) is to wait for the termination of any tasks dependent on the master.] {*blocked (waiting for dependents to terminate)* [partial]} The task executing the master is blocked until all the dependents have terminated. [Any remaining finalization is then performed and the master is left.] 5

6/1 Completion of a task (and the corresponding task_body) can occur when the task is blocked at a select_-statement with an open terminate_alternative (see 9.7.1); the open terminate_alternative is selected if and only if the following conditions are satisfied:

7/2 • {*AI95-00415-01*} The task depends on some completed master; and

8 • Each task that depends on the master considered is either already terminated or similarly blocked at a select_statement with an open terminate_alternative.

9 When both conditions are satisfied, the task considered becomes completed, together with all tasks that depend on the master considered that are not yet completed.

9.a **Ramification:** Any required finalization is performed after the selection of terminate_alternatives. The tasks are not callable during the finalization. In some ways it is as though they were aborted.

NOTES

10 8 The full view of a limited private type can be a task type, or can have subcomponents of a task type. Creation of an object of such a type creates dependences according to the full type.

11 9 An object_renaming_declaration defines a new view of an existing entity and hence creates no further dependence.

12 10 The rules given for the collective completion of a group of tasks all blocked on select_statements with open terminate_alternatives ensure that the collective completion can occur only when there are no remaining active tasks that could call one of the tasks being collectively completed.

13 11 If two or more tasks are blocked on select_statements with open terminate_alternatives, and become completed collectively, their finalization actions proceed concurrently.

14 12 The completion of a task can occur due to any of the following:

15 • the raising of an exception during the elaboration of the declarative_part of the corresponding task_body;

16 • the completion of the handled_sequence_of_statements of the corresponding task_body;

17 • the selection of an open terminate_alternative of a select_statement in the corresponding task_body;

18 • the abort of the task.

Examples

19 *Example of task dependence:*

20
```
declare
   type Global is access Server;        -- see 9.1
   A, B : Server;
   G    : Global;
begin
   -- activation of A and B
   declare
      type Local is access Server;
      X : Global := new Server;  -- activation of X.all
      L : Local  := new Server;  -- activation of L.all
      C : Server;
   begin
      -- activation of C
      G := X;  -- both G and X designate the same task object
      ...
   end;  -- await termination of C and L.all (but not X.all)
   ...
end;  -- await termination of A, B, and G.all
```

Wording Changes from Ada 83

20.a We have revised the wording to be consistent with the definition of master now given in 7.6.1, “Completion and Finalization”.

20.b Tasks that used to depend on library packages in Ada 83, now depend on the (implicit) task_body of the environment task (see 10.2). Therefore, the environment task has to wait for them before performing library level finalization and terminating the partition. In Ada 83 the requirement to wait for tasks that depended on library packages was not as clear.

What was "collective termination" is now "collective completion" resulting from selecting terminate_alternatives. This is because finalization still occurs for such tasks, and this happens after selecting the terminate_alternative, but before termination. 20.c

Wording Changes from Ada 95

{*AI95-00416-01*} Added missing wording that explained the master of tasks that are neither object declarations nor allocators, such as function returns. 20.d/2

9.4 Protected Units and Protected Objects

{*protected object*} {*protected operation*} {*protected subprogram*} {*protected entry*} A *protected object* provides coordinated access to shared data, through calls on its visible *protected operations*, which can be *protected subprograms* or *protected entries*. {*protected declaration*} {*protected unit*} {*protected declaration*} A *protected unit* is declared by a *protected declaration*, which has a corresponding protected_body. A protected declaration may be a protected_type_declaration, in which case it declares a named protected type; alternatively, it may be a single_protected_declaration, in which case it defines an anonymous protected type, as well as declaring a named protected object of that type. {*broadcast signal: See protected object*} 1

Syntax

{*AI95-00345-01*} protected_type_declaration ::= 2/2
  protected type defining_identifier [known_discriminant_part] **is**
    [**new** interface_list **with**]
    protected_definition;

{*AI95-00399-01*} single_protected_declaration ::= 3/2
  protected defining_identifier **is**
    [**new** interface_list **with**]
    protected_definition;

protected_definition ::= 4
    { protected_operation_declaration }
[**private**
    { protected_element_declaration }]
  end [*protected*_identifier]

{*8652/0009*} {*AI95-00137-01*} protected_operation_declaration ::= subprogram_declaration 5/1
    | entry_declaration
    | aspect_clause

protected_element_declaration ::= protected_operation_declaration 6
    | component_declaration

Reason: We allow the operations and components to be mixed because that's how other things work (for example, package declarations). We have relaxed the ordering rules for the items inside declarative_parts and task_definitions as well. 6.a

protected_body ::= 7
  protected body defining_identifier **is**
   { protected_operation_item }
  end [*protected*_identifier];

{*8652/0009*} {*AI95-00137-01*} protected_operation_item ::= subprogram_declaration 8/1
    | subprogram_body
    | entry_body
    | aspect_clause

9 If a *protected*_identifier appears at the end of a protected_definition or protected_body, it shall repeat the defining_identifier.

Legality Rules

10/2 *This paragraph was deleted.*{*AI95-00345-01*}

10.a/2 *This paragraph was deleted.*

Static Semantics

11/2 {*AI95-00345-01*} {*AI95-00401-01*} A protected_definition defines a protected type and its first subtype. {*visible part (of a protected unit)* [partial]} The list of protected_operation_declarations of a protected_definition, together with the known_discriminant_part, if any, is called the visible part of the protected unit. [{*private part (of a protected unit)* [partial]} The optional list of protected_element_declarations after the reserved word **private** is called the private part of the protected unit.]

11.a **Proof:** Private part is defined in Section 8.

11.1/2 {*AI95-00345-01*} {*AI95-00397-01*} {*AI95-00399-01*} {*AI95-00419-01*} For a protected declaration with an interface_list, the protected type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the protected type or an access parameter designating the protected type, and there is a protected_operation_declaration for a protected subprogram or single entry with the same identifier within the protected declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming protected subprogram or entry.{*implemented (by a protected subprogram)* [partial]} {*implemented (by a protected entry)* [partial]} {*type conformance (required)*}

11.b/2 **Ramification:** The inherited subprograms can only come from an interface given as part of the protected declaration.

Legality Rules

11.2/2 {*AI95-00345-01*} {*requires a completion (protected_declaration})* [partial]} A protected declaration requires a completion[, which shall be a protected_body,] and every protected_body shall be the completion of some protected declaration.

11.c/2 **To be honest:** The completion can be a pragma Import, if the implementation supports it.

11.3/2 {*AI95-00345-01*} {*AI95-00399-01*} [Each *interface*_subtype_mark of an interface_list appearing within a protected declaration shall denote a limited interface type that is not a task interface.]

11.d/2 **Proof:** 3.9.4 requires that an interface_list only name interface types, and limits the descendants of the various kinds of interface types. Only a limited, protected, or synchronized interface can have a protected type descendant. Nonlimited or task interfaces are not allowed, as they offer operations that a protected type does not have.

11.4/2 {*AI95-00397-01*} The prefixed view profile of an explicitly declared primitive subprogram of a tagged protected type shall not be type conformant with any protected operation of the protected type, if the first parameter of the subprogram is of the protected type or is an access parameter designating the protected type.{*type conformance (required)*}

11.e/2 **Reason:** This prevents the existence of two operations with the same name and profile which could be called with a prefixed view. If the operation was inherited, this would be illegal by the following rules; this rule puts inherited and non-inherited routines on the same footing. Note that this only applies to tagged protected types (that is, those with an interface in their declaration); we do that as there is no problem with prefixed view calls of primitive operations for "normal" protected types, and having this rule apply to all protected types would be incompatible with Ada 95.

11.5/2 {*AI95-00345-01*} {*AI95-00399-01*} For each primitive subprogram inherited by the type declared by a protected declaration, at most one of the following shall apply:

- {*AI95-00345-01*} the inherited subprogram is overridden with a primitive subprogram of the protected type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or{*subtype conformance (required)*} 11.6/2
- {*AI95-00345-01*} {*AI95-00397-01*} the inherited subprogram is implemented by a protected subprogram or single entry of the protected type, in which case its prefixed view profile shall be subtype conformant with that of the protected subprogram or entry. {*subtype conformance (required)*} 11.7/2

If neither applies, the inherited subprogram shall be a null procedure. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 11.8/2

> **Reason:** Each inherited subprogram can only have a single implementation (either from overriding a subprogram, implementing a subprogram, or implementing an entry), and must have an implementation unless the subprogram is a null procedure. 11.f/2

{*AI95-00345-01*} If an inherited subprogram is implemented by a protected procedure or an entry, then the first parameter of the inherited subprogram shall be of mode **out** or **in out**, or an access-to-variable parameter. 11.9/2

> **Reason:** For a protected procedure or entry, the protected object can be read or written (see 9.5.1}. A subprogram that is implemented by a protected procedure or entry must have a profile which reflects that in order to avoid confusion. 11.g/2

{*AI95-00397-01*} If a protected subprogram declaration has an overriding_indicator, then at the point of the declaration: 11.10/2

- if the overriding_indicator is **overriding**, then the subprogram shall implement an inherited subprogram; 11.11/2
- if the overriding_indicator is **not overriding**, then the subprogram shall not implement any inherited subprogram. 11.12/2

{*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 11.13/2

> **Discussion:** These rules are subtly different than those for subprograms (see 8.3.1) because there cannot be “late” inheritance of primitives from interfaces. Hidden (that is, private) interfaces are prohibited explicitly (see 7.3), as are hidden primitive operations (as private operations of public abstract types are prohibited — see 3.9.3). 11.h/2

Dynamic Semantics

[{*elaboration (protected declaration)* [partial]} The elaboration of a protected declaration elaborates the protected_definition. {*elaboration (single_protected_declaration)* [partial]} The elaboration of a single_protected_declaration also creates an object of an (anonymous) protected type.] 12

> **Proof:** This is redundant with the general rules for the elaboration of a full_type_declaration and an object_declaration. 12.a

{*elaboration (protected_definition)* [partial]} [The elaboration of a protected_definition creates the protected type and its first subtype;] it also includes the elaboration of the component_declarations and protected_operation_declarations in the given order. 13

[{*initialization (of a protected object)* [partial]} As part of the initialization of a protected object, any per-object constraints (see 3.8) are elaborated.] 14

> **Discussion:** We do not mention pragmas since each pragma has its own elaboration rules. 14.a

{*elaboration (protected_body)* [partial]} The elaboration of a protected_body has no other effect than to establish that protected operations of the type can from then on be called without failing the Elaboration_Check. 15

16 The content of an object of a given protected type includes:

17 • The values of the components of the protected object, including (implicitly) an entry queue for each entry declared for the protected object;

17.a **Ramification:** "For each entry" implies one queue for each single entry, plus one for each entry of each entry family.

18 • {*execution resource (associated with a protected object)* [partial]} A representation of the state of the execution resource *associated* with the protected object (one such resource is associated with each protected object).

19 [The execution resource associated with a protected object has to be acquired to read or update any components of the protected object; it can be acquired (as part of a protected action — see 9.5.1) either for concurrent read-only access, or for exclusive read-write access.]

20 {*finalization (of a protected object)* [partial]} {*Program_Error (raised by failure of run-time check)*} As the first step of the *finalization* of a protected object, each call remaining on any entry queue of the object is removed from its queue and Program_Error is raised at the place of the corresponding entry_call_statement.

20.a **Reason:** This is analogous to the raising of Tasking_Error in callers of a task that completes before accepting the calls. This situation can only occur due to a requeue (ignoring premature unchecked_deallocation), since any task that has accessibility to a protected object is awaited before finalizing the protected object. For example:

20.b
```
procedure Main is
    task T is
        entry E;
    end T;
```

20.c
```
    task body T is
        protected PO is
            entry Ee;
        end PO;
```

20.d
```
        protected body PO is
            entry Ee when False is
            begin
                null;
            end Ee;
        end PO;
    begin
        accept E do
            requeue PO.Ee;
        end E;
    end T;
begin
    T.E;
end Main;
```

20.e The environment task is queued on PO.EE when PO is finalized.

20.f In a real example, a server task might park callers on a local protected object for some useful purpose, so we didn't want to disallow this case.

Bounded (Run-Time) Errors

20.1/2 {*AI95-00280-01*} {*bounded error (cause)* [partial]} It is a bounded error to call an entry or subprogram of a protected object after that object is finalized. If the error is detected, Program_Error is raised. Otherwise, the call proceeds normally, which may leave a task queued forever.

20.g/2 **Reason:** This is very similar to the finalization rule. It is a bounded error so that an implementation can avoid the overhead of the check if it can ensure that the call still will operate properly. Such an implementation cannot need to return resources (such as locks) to an executive that it needs to execute calls.

20.h/2 This case can happen (and has happened in production code) when a protected object is accessed from the Finalize routine of a type. For example:

20.i/2
```
with Ada.Finalization.Controlled;
package Window_Manager is
    ...
    type Root_Window is new Ada.Finalization.Controlled with private;
    type Any_Window is access all Root_Window;
    ...
private
    ...
    procedure Finalize (Object : in out Root_Window);
    ...
end Window_Manager;
```

20.j/2
```
package body Window_Manager is
   protected type Lock is
       entry Get_Lock;
       procedure Free_Lock;
   ...
   end Lock;
```

20.k/2
```
   Window_Lock : Lock;
```

20.l/2
```
   procedure Finalize (Object : in out Root_Window) is
   begin
       Window_Lock.Get_Lock;
       ...
       Window_Lock.Free_Lock;
   end Finalize;
   ...
   A_Window : Any_Window := new Root_Window;
end Window_Manager;
```

The environment task will call Window_Lock for the object allocated for A_Window when the collection for Any_Window is finalized, which will happen after the finalization of Window_Lock (because finalization of the package body will occur before that of the package specification). 20.m/2

NOTES

13 {*AI95-00382-01*} Within the declaration or body of a protected unit other than in an access_definition, the name of the protected unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding protected type (and thus the name cannot be used as a subtype_mark). 21/2

Discussion: {*AI95-00382-01*} It can be used as a subtype_mark in an anonymous access type. In addition, it is possible to refer to some other subtype of the protected type within its body, presuming such a subtype has been declared between the protected_type_declaration and the protected_body. 21.a/2

14 A selected_component can be used to denote a discriminant of a protected object (see 4.1.3). Within a protected unit, the name of a discriminant of the protected type denotes the corresponding discriminant of the current instance of the unit. 22

15 {*AI95-00287-01*} A protected type is a limited type (see 7.5), and hence precludes use of assignment_statements and predefined equality operators. 23/2

16 The bodies of the protected operations given in the protected_body define the actions that take place upon calls to the protected operations. 24

17 The declarations in the private part are only visible within the private part and the body of the protected unit. 25

Reason: Component_declarations are disallowed in a protected_body because, for efficiency, we wish to allow the compiler to determine the size of protected objects (when not dynamic); the compiler cannot necessarily see the body. Furthermore, the semantics of initialization of such objects would be problematic — we do not wish to give protected objects complex initialization semantics similar to task activation. 25.a

The same applies to entry_declarations, since an entry involves an implicit component — the entry queue. 25.b

Examples

Example of declaration of protected type and corresponding body: 26

27
```
protected type Resource is
   entry Seize;
   procedure Release;
private
   Busy : Boolean := False;
end Resource;
```

28
```
protected body Resource is
   entry Seize when not Busy is
   begin
      Busy := True;
   end Seize;
```

29
```
   procedure Release is
   begin
      Busy := False;
   end Release;
end Resource;
```

30 *Example of a single protected declaration and corresponding body:*

31
```
protected Shared_Array is
   --  Index, Item, and Item_Array are global types
   function  Component    (N : in Index) return Item;
   procedure Set_Component(N : in Index; E : in  Item);
private
   Table : Item_Array(Index) := (others => Null_Item);
end Shared_Array;
```

32
```
protected body Shared_Array is
   function Component(N : in Index) return Item is
   begin
      return Table(N);
   end Component;
```

33
```
   procedure Set_Component(N : in Index; E : in Item) is
   begin
      Table(N) := E;
   end Set_Component;
end Shared_Array;
```

34 *Examples of protected objects:*

35
```
Control  : Resource;
Flags    : array(1 .. 100) of Resource;
```

Extensions to Ada 83

35.a {*extensions to Ada 83*} This entire clause is new; protected units do not exist in Ada 83.

Extensions to Ada 95

35.b/2 {*AI95-00345-01*} {*AI95-00397-01*} {*AI95-00399-01*} {*AI95-00401-01*} {*AI95-00419-01*} {*extensions to Ada 95*} Protected types and single protected objects can be derived from one or more interfaces. Operations declared in the protected type can implement the primitive operations of an interface. Overriding_indicators can be used to specify whether or not a protected operation implements a primitive operation.

Wording Changes from Ada 95

35.c/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Changed representation clauses to aspect clauses to reflect that they are used for more than just representation.

35.d/2 {*AI95-00280-01*} Described what happens when an operation of a finalized protected object is called.

35.e/2 {*AI95-00287-01*} Revised the note on operations of protected types to reflect that limited types do have an assignment operation, but not copying (assignment_statements).

35.f/2 {*AI95-00382-01*} Revised the note on use of the name of a protected type within itself to reflect the exception for anonymous access types.

9.5 Intertask Communication

1 {*intertask communication*} {*critical section: See intertask communication*} The primary means for intertask communication is provided by calls on entries and protected subprograms. Calls on protected subprograms allow coordinated access to shared data objects. Entry calls allow for blocking the caller

until a given condition is satisfied (namely, that the corresponding entry is open — see 9.5.3), and then communicating data or control information directly with another task or indirectly via a shared protected object.

Static Semantics

{*target object (of a call on an entry or a protected subprogram)*} Any call on an entry or on a protected subprogram identifies a *target object* for the operation, which is either a task (for an entry call) or a protected object (for an entry call or a protected subprogram call). The target object is considered an implicit parameter to the operation, and is determined by the operation name (or prefix) used in the call on the operation, as follows: 2

- If it is a direct_name or expanded name that denotes the declaration (or body) of the operation, then the target object is implicitly specified to be the current instance of the task or protected unit immediately enclosing the operation; {*internal call*} such a call is defined to be an *internal call*; 3
- If it is a selected_component that is not an expanded name, then the target object is explicitly specified to be the task or protected object denoted by the prefix of the name; {*external call*} such a call is defined to be an *external call*; 4

Discussion: For example: 4.a

4.b
```
protected type Pt is
  procedure Op1;
  procedure Op2;
end Pt;
```

4.c
```
PO : Pt;
Other_Object : Some_Other_Protected_Type;
```

4.d
```
protected body Pt is
  procedure Op1 is begin ... end Op1;
```

4.e
```
  procedure Op2 is
  begin
    Op1; -- An internal call.
    Pt.Op1; -- Another internal call.
    PO.Op1; -- An external call. It the current instance is PO, then
            -- this is a bounded error (see 9.5.1).
    Other_Object.Some_Op; -- An external call.
  end Op2;
end Pt;
```

- If the name or prefix is a dereference (implicit or explicit) of an access-to-protected-subprogram value, then the target object is determined by the prefix of the Access attribute_reference that produced the access value originally, and the call is defined to be an *external call*; 5
- If the name or prefix denotes a subprogram_renaming_declaration, then the target object is as determined by the name of the renamed entity. 6

{*target object (of a requeue_statement)*} {*internal requeue*} {*external requeue*} A corresponding definition of target object applies to a requeue_statement (see 9.5.4), with a corresponding distinction between an *internal requeue* and an *external requeue*. 7

Legality Rules

{*AI95-00345-01*} The view of the target protected object associated with a call of a protected procedure or entry shall be a variable. 7.1/2

Dynamic Semantics

Within the body of a protected operation, the current instance (see 8.6) of the immediately enclosing protected unit is determined by the target object specified (implicitly or explicitly) in the call (or requeue) on the protected operation. 8

8.a **To be honest:** The current instance is defined in the same way within the body of a subprogram declared immediately within a protected_body.

9 Any call on a protected procedure or entry of a target protected object is defined to be an update to the object, as is a requeue on such an entry.

9.a **Reason:** Read/write access to the components of a protected object is granted while inside the body of a protected procedure or entry. Also, any protected entry call can change the value of the Count attribute, which represents an update. Any protected procedure call can result in servicing the entries, which again might change the value of a Count attribute.

Wording Changes from Ada 95

9.b/2 {*AI95-00345-01*} Added a Legality Rule to make it crystal-clear that the protected object of an entry or procedure call must be a variable. This rule was implied by the Dynamic Semantics here, along with the Static Semantics of 3.3, but it is much better to explicitly say it. While many implementations have gotten this wrong, this is not an incompatibility — allowing updates of protected constants has always been wrong.

9.5.1 Protected Subprograms and Protected Actions

1 {*protected subprogram*} {*protected procedure*} {*protected function*} A *protected subprogram* is a subprogram declared immediately within a protected_definition. Protected procedures provide exclusive read-write access to the data of a protected object; protected functions provide concurrent read-only access to the data.

1.a **Ramification:** A subprogram declared immediately within a protected_body is not a protected subprogram; it is an intrinsic subprogram. See 6.3.1, “Conformance Rules”.

Static Semantics

2 Within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit is defined to be a constant [(that is, its subcomponents may be read but not updated)]. Within the body of a protected procedure (or a procedure declared immediately within a protected_body), and within an entry_body, the current instance is defined to be a variable [(updating is permitted)].

2.a **Ramification:** The current instance is like an implicit parameter, of mode **in** for a protected function, and of mode **in out** for a protected procedure (or protected entry).

Dynamic Semantics

3 {*execution (protected subprogram call)* [partial]} For the execution of a call on a protected subprogram, the evaluation of the name or prefix and of the parameter associations, and any assigning back of **in out** or **out** parameters, proceeds as for a normal subprogram call (see 6.4). If the call is an internal call (see 9.5), the body of the subprogram is executed as for a normal subprogram call. If the call is an external call, then the body of the subprogram is executed as part of a new *protected action* on the target protected object; the protected action completes after the body of the subprogram is executed. [A protected action can also be started by an entry call (see 9.5.3).]

4 {*protected action*} A new protected action is not started on a protected object while another protected action on the same protected object is underway, unless both actions are the result of a call on a protected function. This rule is expressible in terms of the execution resource associated with the protected object:

5 • {*protected action (start)*} {*acquire (execution resource associated with protected object)*} *Starting* a protected action on a protected object corresponds to *acquiring* the execution resource associated with the protected object, either for concurrent read-only access if the protected action is for a call on a protected function, or for exclusive read-write access otherwise;

6 • {*protected action (complete)*} {*release (execution resource associated with protected object)*} *Completing* the protected action corresponds to *releasing* the associated execution resource.

[After performing an operation on a protected object other than a call on a protected function, but prior to completing the associated protected action, the entry queues (if any) of the protected object are serviced (see 9.5.3).] 7

Bounded (Run-Time) Errors

{*bounded error (cause)* [partial]} During a protected action, it is a bounded error to invoke an operation that is *potentially blocking*. {*potentially blocking operation*} {*blocking, potentially*} The following are defined to be potentially blocking operations: 8

> **Reason:** Some of these operations are not directly blocking. However, they are still treated as bounded errors during a protected action, because allowing them might impose an undesirable implementation burden. 8.a

- a select_statement; 9
- an accept_statement; 10
- an entry_call_statement; 11
- a delay_statement; 12
- an abort_statement; 13
- task creation or activation; 14
- an external call on a protected subprogram (or an external requeue) with the same target object as that of the protected action; 15

> **Reason:** This is really a deadlocking call, rather than a blocking call, but we include it in this list for simplicity. 15.a

- a call on a subprogram whose body contains a potentially blocking operation. 16

> **Reason:** This allows an implementation to check and raise Program_Error as soon as a subprogram is called, rather than waiting to find out whether it actually reaches the potentially blocking operation. This in turn allows the potentially blocking operation check to be performed prior to run time in some environments. 16.a

{*Program_Error (raised by failure of run-time check)*} If the bounded error is detected, Program_Error is raised. If not detected, the bounded error might result in deadlock or a (nested) protected action on the same target object. 17

> **Discussion:** {*AI95-00305-01*} By "nested protected action", we mean that an additional protected action can be started by another task on the same protected object. This means that mutual exclusion may be broken in this bounded error case. A way to ensure that this does not happen is to use pragma Detect_Blocking (see H.5). 17.a/2

Certain language-defined subprograms are potentially blocking. In particular, the subprograms of the language-defined input-output packages that manipulate files (implicitly or explicitly) are potentially blocking. Other potentially blocking subprograms are identified where they are defined. When not specified as potentially blocking, a language-defined subprogram is nonblocking. 18

> **Discussion:** {*AI95-00178-01*} Any subprogram in a language-defined input-output package that has a file parameter or result or operates on a default file is considered to manipulate a file. An instance of a language-defined input-output generic package provides subprograms that are covered by this rule. The only subprograms in language-defined input-output packages not covered by this rule (and thus not potentially blocking) are the Get and Put routines that take string parameters defined in the packages nested in Text_IO. 18.a/2

NOTES

18 If two tasks both try to start a protected action on a protected object, and at most one is calling a protected function, then only one of the tasks can proceed. Although the other task cannot proceed, it is not considered blocked, and it might be consuming processing resources while it awaits its turn. There is no language-defined ordering or queuing presumed for tasks competing to start a protected action — on a multiprocessor such tasks might use busy-waiting; for monoprocessor considerations, see D.3, "Priority Ceiling Locking". 19

> **Discussion:** The intended implementation on a multi-processor is in terms of "spin locks" — the waiting task will spin. 19.a

19 The body of a protected unit may contain declarations and bodies for local subprograms. These are not visible outside the protected unit. 20

21 20 The body of a protected function can contain internal calls on other protected functions, but not protected procedures, because the current instance is a constant. On the other hand, the body of a protected procedure can contain internal calls on both protected functions and procedures.

22 21 From within a protected action, an internal call on a protected subprogram, or an external call on a protected subprogram with a different target object is not considered a potentially blocking operation.

22.a **Reason:** This is because a task is not considered blocked while attempting to acquire the execution resource associated with a protected object. The acquisition of such a resource is rather considered part of the normal competition for execution resources between the various tasks that are ready. External calls that turn out to be on the same target object are considered potentially blocking, since they can deadlock the task indefinitely.

22.1/2 22 {*AI95-00305-01*} The pragma Detect_Blocking may be used to ensure that all executions of potentially blocking operations during a protected action raise Program_Error. See H.5.

Examples

23 *Examples of protected subprogram calls (see 9.4):*

24
```
Shared_Array.Set_Component(N, E);
E := Shared_Array.Component(M);
Control.Release;
```

Wording Changes from Ada 95

24.a/2 {*AI95-00305-01*} Added a note pointing out the existence of pragma Detect_Blocking. This pragma can be used to ensure portable (somewhat pessimistic) behavior of protected actions by converting the Bounded Error into a required check.

9.5.2 Entries and Accept Statements

1 Entry_declarations, with the corresponding entry_bodies or accept_statements, are used to define potentially queued operations on tasks and protected objects.

Syntax

2/2 {*AI95-00397-01*} entry_declaration ::=
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile;

3 accept_statement ::=
  accept *entry*_direct_name [(entry_index)] parameter_profile [**do**
    handled_sequence_of_statements
  end [*entry*_identifier]];

3.a **Reason:** We cannot use defining_identifier for accept_statements. Although an accept_statement is sort of like a body, it can appear nested within a block_statement, and therefore be hidden from its own entry by an outer homograph.

4 entry_index ::= expression

5 entry_body ::=
  entry defining_identifier entry_body_formal_part entry_barrier **is**
    declarative_part
  begin
    handled_sequence_of_statements
  end [*entry*_identifier];

5.a/2 **Discussion:** {*AI95-00397-01*} We don't allow an overriding_indicator on an entry_body because entries always implement procedures at the point of the type declaration; there is no late implementation. And we don't want to have to think about overriding_indicators on accept_statements.

6 entry_body_formal_part ::= [(entry_index_specification)] parameter_profile

7 entry_barrier ::= **when** condition

8 entry_index_specification ::= **for** defining_identifier **in** discrete_subtype_definition

If an *entry*_identifier appears at the end of an accept_statement, it shall repeat the *entry*_direct_name. If an *entry*_identifier appears at the end of an entry_body, it shall repeat the defining_identifier. 9

[An entry_declaration is allowed only in a protected or task declaration.] 10

Proof: This follows from the BNF. 10.a

{*AI95-00397-01*} An overriding_indicator is not allowed in an entry_declaration that includes a discrete_subtype_definition. 10.1/2

Reason: An entry family can never implement something, so allowing an indicator is felt by the majority of the ARG to be redundant. 10.a.1/2

Name Resolution Rules

{*expected profile (accept_statement entry_direct_name)* [partial]} In an accept_statement, the expected profile for the *entry*_direct_name is that of the entry_declaration; {*expected type (entry_index)* [partial]} the expected type for an entry_index is that of the subtype defined by the discrete_subtype_definition of the corresponding entry_declaration. 11

Within the handled_sequence_of_statements of an accept_statement, if a selected_component has a prefix that denotes the corresponding entry_declaration, then the entity denoted by the prefix is the accept_statement, and the selected_component is interpreted as an expanded name (see 4.1.3)[; the selector_name of the selected_component has to be the identifier for some formal parameter of the accept_statement]. 12

Proof: The only declarations that occur immediately within the declarative region of an accept_statement are those for its formal parameters. 12.a

Legality Rules

An entry_declaration in a task declaration shall not contain a specification for an access parameter (see 3.10). 13

Reason: Access parameters for task entries would require a complex implementation. For example: 13.a

13.b
```
task T is
   entry E(Z : access Integer); -- Illegal!
end T;
```

13.c
```
task body T is
begin
   declare
      type A is access all Integer;
      X : A;
      Int : aliased Integer;
      task Inner;
      task body Inner is
      begin
         T.E(Int'Access);
      end Inner;
   begin
      accept E(Z : access Integer) do
         X := A(Z); -- Accessibility_Check
      end E;
   end;
end T;
```

Implementing the Accessibility_Check inside the accept_statement for E is difficult, since one does not know whether the entry caller is calling from inside the immediately enclosing declare block or from outside it. This means that the lexical nesting level associated with the designated object is not sufficient to determine whether the Accessibility_Check should pass or fail. 13.d

Note that such problems do not arise with protected entries, because entry_bodies are always nested immediately within the protected_body; they cannot be further nested as can accept_statements, nor can they be called from within the protected_body (since no entry calls are permitted inside a protected_body). 13.e

13.1/2 {*AI95-00397-01*} If an entry_declaration has an overriding_indicator, then at the point of the declaration:

13.2/2
- if the overriding_indicator is **overriding**, then the entry shall implement an inherited subprogram;

13.3/2
- if the overriding_indicator is **not overriding**, then the entry shall not implement any inherited subprogram.

13.4/2 {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

13.f/2 **Discussion:** These rules are subtly different than those for subprograms (see 8.3.1) because there cannot be "late" inheritance of primitives from interfaces. Hidden (that is, private) interfaces are prohibited explicitly (see 7.3), as are hidden primitive operations (as private operations of public abstract types are prohibited — see 3.9.3).

14 For an accept_statement, the innermost enclosing body shall be a task_body, and the *entry*_direct_name shall denote an entry_declaration in the corresponding task declaration; the profile of the accept_statement shall conform fully to that of the corresponding entry_declaration. {*full conformance (required)*} An accept_statement shall have a parenthesized entry_index if and only if the corresponding entry_declaration has a discrete_subtype_definition.

15 An accept_statement shall not be within another accept_statement that corresponds to the same entry_declaration, nor within an asynchronous_select inner to the enclosing task_body.

15.a **Reason:** Accept_statements are required to be immediately within the enclosing task_body (as opposed to being in a nested subprogram) to ensure that a nested task does not attempt to accept the entry of its enclosing task. We considered relaxing this restriction, either by making the check a run-time check, or by allowing a nested task to accept an entry of its enclosing task. However, neither change seemed to provide sufficient benefit to justify the additional implementation burden.

15.b Nested accept_statements for the same entry (or entry family) are prohibited to ensure that there is no ambiguity in the resolution of an expanded name for a formal parameter of the entry. This could be relaxed by allowing the inner one to hide the outer one from all visibility, but again the small added benefit didn't seem to justify making the change for Ada 95.

15.c Accept_statements are not permitted within asynchronous_select statements to simplify the semantics and implementation: an accept_statement in an abortable_part could result in Tasking_Error being propagated from an entry call even though the target task was still callable; implementations that use multiple tasks implicitly to implement an asynchronous_select might have trouble supporting "up-level" accepts. Furthermore, if accept_statements were permitted in the abortable_part, a task could call its own entry and then accept it in the abortable_part, leading to rather unusual and possibly difficult-to-specify semantics.

16 {*requires a completion (protected entry_declaration)* [partial]} An entry_declaration of a protected unit requires a completion[, which shall be an entry_body,] {*only as a completion (entry_body)* [partial]} and every entry_body shall be the completion of an entry_declaration of a protected unit. {*completion legality (entry_body)* [partial]} The profile of the entry_body shall conform fully to that of the corresponding declaration. {*full conformance (required)*}

16.a **Ramification:** An entry_declaration, unlike a subprogram_declaration, cannot be completed with a renaming_declaration.

16.b **To be honest:** The completion can be a pragma Import, if the implementation supports it.

16.c **Discussion:** The above applies only to protected entries, which are the only ones completed with entry_bodies. Task entries have corresponding accept_statements instead of having entry_bodies, and we do not consider an accept_statement to be a "completion," because a task entry_declaration is allowed to have zero, one, or more than one corresponding accept_statements.

17 An entry_body_formal_part shall have an entry_index_specification if and only if the corresponding entry_declaration has a discrete_subtype_definition. In this case, the discrete_subtype_definitions of the entry_declaration and the entry_index_specification shall fully conform to one another (see 6.3.1). {*full conformance (required)*}

A name that denotes a formal parameter of an entry_body is not allowed within the entry_barrier of the entry_body. 18

Static Semantics

The parameter modes defined for parameters in the parameter_profile of an entry_declaration are the same as for a subprogram_declaration and have the same meaning (see 6.2). 19

> **Discussion:** Note that access parameters are not allowed for task entries (see above). 19.a

{*family (entry)*} {*entry family*} {*entry index subtype*} An entry_declaration with a discrete_subtype_definition (see 3.6) declares a *family* of distinct entries having the same profile, with one such entry for each value of the *entry index subtype* defined by the discrete_subtype_definition. [A name for an entry of a family takes the form of an indexed_component, where the prefix denotes the entry_declaration for the family, and the index value identifies the entry within the family.] {*single entry*} {*entry (single)*} The term *single entry* is used to refer to any entry other than an entry of an entry family. 20

In the entry_body for an entry family, the entry_index_specification declares a named constant whose subtype is the entry index subtype defined by the corresponding entry_declaration; {*named entry index*} the value of the *named entry index* identifies which entry of the family was called. 21

> **Ramification:** The discrete_subtype_definition of the entry_index_specification is not elaborated; the subtype of the named constant declared is defined by the discrete_subtype_definition of the corresponding entry_declaration, which is elaborated, either when the type is declared, or when the object is created, if its constraint is per-object. 21.a

Dynamic Semantics

{*8652/0002*} {*AI95-00171-01*} {*elaboration (entry_declaration)* [partial]} The elaboration of an entry_declaration for an entry family consists of the elaboration of the discrete_subtype_definition, as described in 3.8. The elaboration of an entry_declaration for a single entry has no effect. 22/1

> **Discussion:** The elaboration of the declaration of a protected subprogram has no effect, as specified in clause 6.1. The default initialization of an object of a task or protected type is covered in 3.3.1. 22.a

[The actions to be performed when an entry is called are specified by the corresponding accept_statements (if any) for an entry of a task unit, and by the corresponding entry_body for an entry of a protected unit.] 23

{*execution (accept_statement)* [partial]} For the execution of an accept_statement, the entry_index, if any, is first evaluated and converted to the entry index subtype; this index value identifies which entry of the family is to be accepted. {*implicit subtype conversion (entry index)* [partial]} {*blocked (on an accept_statement)* [partial]} {*selection (of an entry caller)*} Further execution of the accept_statement is then blocked until a caller of the corresponding entry is selected (see 9.5.3), whereupon the handled_sequence_of_statements, if any, of the accept_statement is executed, with the formal parameters associated with the corresponding actual parameters of the selected entry call. Upon completion of the handled_sequence_of_statements, the accept_statement completes and is left. When an exception is propagated from the handled_sequence_of_statements of an accept_statement, the same exception is also raised by the execution of the corresponding entry_call_statement. 24

> **Ramification:** This is in addition to propagating it to the construct containing the accept_statement. In other words, for a rendezvous, the raising splits in two, and continues concurrently in both tasks. 24.a
>
> The caller gets a new occurrence; this isn't considered propagation. 24.b
>
> Note that we say “propagated from the handled_sequence_of_statements of an accept_statement”, not “propagated from an accept_statement.” The latter would be wrong — we don't want exceptions propagated by the entry_index to be sent to the caller (there is none yet!). 24.c

{*rendezvous*} The above interaction between a calling task and an accepting task is called a *rendezvous*. [After a rendezvous, the two tasks continue their execution independently.] 25

26 [An entry_body is executed when the condition of the entry_barrier evaluates to True and a caller of the corresponding single entry, or entry of the corresponding entry family, has been selected (see 9.5.3).] {*execution (entry_body)* [partial]} For the execution of the entry_body, the declarative_part of the entry_-body is elaborated, and the handled_sequence_of_statements of the body is executed, as for the execution of a subprogram_body. The value of the named entry index, if any, is determined by the value of the entry index specified in the *entry*_name of the selected entry call (or intermediate requeue_-statement — see 9.5.4).

26.a **To be honest:** If the entry had been renamed as a subprogram, and the call was a procedure_call_statement using the name declared by the renaming, the entry index (if any) comes from the entry name specified in the subprogram_renaming_declaration.

NOTES

27 23 A task entry has corresponding accept_statements (zero or more), whereas a protected entry has a corresponding entry_body (exactly one).

28 24 A consequence of the rule regarding the allowed placements of accept_statements is that a task can execute accept_statements only for its own entries.

29/2 25 {*AI95-00318-02*} A return statement (see 6.5) or a requeue_statement (see 9.5.4) may be used to complete the execution of an accept_statement or an entry_body.

29.a **Ramification:** An accept_statement need not have a handled_sequence_of_statements even if the corresponding entry has parameters. Equally, it can have a handled_sequence_of_statements even if the corresponding entry has no parameters.

29.b **Ramification:** A single entry overloads a subprogram, an enumeration literal, or another single entry if they have the same defining_identifier. Overloading is not allowed for entry family names. A single entry or an entry of an entry family can be renamed as a procedure as explained in 8.5.4.

30 26 The condition in the entry_barrier may reference anything visible except the formal parameters of the entry. This includes the entry index (if any), the components (including discriminants) of the protected object, the Count attribute of an entry of that protected object, and data global to the protected unit.

31 The restriction against referencing the formal parameters within an entry_barrier ensures that all calls of the same entry see the same barrier value. If it is necessary to look at the parameters of an entry call before deciding whether to handle it, the entry_barrier can be "**when** True" and the caller can be requeued (on some private entry) when its parameters indicate that it cannot be handled immediately.

Examples

32 *Examples of entry declarations:*

33
```
entry Read(V : out Item);
entry Seize;
entry Request(Level)(D : Item);  -- a family of entries
```

34 *Examples of accept statements:*

35
```
accept Shut_Down;
```

36
```
accept Read(V : out Item) do
   V := Local_Item;
end Read;
```

37
```
accept Request(Low)(D : Item) do
   ...
end Request;
```

Extensions to Ada 83

37.a {*extensions to Ada 83*} The syntax rule for entry_body is new.

37.b Accept_statements can now have exception_handlers.

Wording Changes from Ada 95

37.c/2 {*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Clarified the elaboration of per-object constraints.

37.d/2 {*AI95-00397-01*} Overriding_indicators can be used on entries; this is only useful when a task or protected type inherits from an interface.

9.5.3 Entry Calls

1 {*entry call*} [An entry_call_statement (an *entry call*) can appear in various contexts.] {*simple entry call*} {*entry call (simple)*} A *simple* entry call is a stand-alone statement that represents an unconditional call on an entry of a target task or a protected object. [Entry calls can also appear as part of select_statements (see 9.7).]

Syntax

2 entry_call_statement ::= *entry*_name [actual_parameter_part];

Name Resolution Rules

3 The *entry*_name given in an entry_call_statement shall resolve to denote an entry. The rules for parameter associations are the same as for subprogram calls (see 6.4 and 6.4.1).

Static Semantics

4 [The *entry*_name of an entry_call_statement specifies (explicitly or implicitly) the target object of the call, the entry or entry family, and the entry index, if any (see 9.5).]

Dynamic Semantics

5 {*open entry*} {*entry (open)*} {*closed entry*} {*entry (closed)*} Under certain circumstances (detailed below), an entry of a task or protected object is checked to see whether it is *open* or *closed*:

- 6 {*open entry (of a task)*} {*closed entry (of a task)*} An entry of a task is open if the task is blocked on an accept_statement that corresponds to the entry (see 9.5.2), or on a selective_accept (see 9.7.1) with an open accept_alternative that corresponds to the entry; otherwise it is closed.
- 7 {*open entry (of a protected object)*} {*closed entry (of a protected object)*} An entry of a protected object is open if the condition of the entry_barrier of the corresponding entry_body evaluates to True; otherwise it is closed. {*Program_Error (raised by failure of run-time check)*} If the evaluation of the condition propagates an exception, the exception Program_Error is propagated to all current callers of all entries of the protected object.

7.a **Reason:** An exception during barrier evaluation is considered essentially a fatal error. All current entry callers are notified with a Program_Error. In a fault-tolerant system, a protected object might provide a Reset protected procedure, or equivalent, to support attempts to restore such a "broken" protected object to a reasonable state.

7.b **Discussion:** Note that the definition of when a task entry is open is based on the state of the (accepting) task, whereas the "openness" of a protected entry is defined only when it is explicitly checked, since the barrier expression needs to be evaluated. Implementation permissions are given (below) to allow implementations to evaluate the barrier expression more or less often than it is checked, but the basic semantic model presumes it is evaluated at the times when it is checked.

8 {*execution (entry_call_statement)* [partial]} For the execution of an entry_call_statement, evaluation of the name and of the parameter associations is as for a subprogram call (see 6.4). {*issue (an entry call)*} The entry call is then *issued*: For a call on an entry of a protected object, a new protected action is started on the object (see 9.5.1). The named entry is checked to see if it is open; {*select an entry call (immediately)*} if open, the entry call is said to be *selected immediately*, and the execution of the call proceeds as follows:

- 9 For a call on an open entry of a task, the accepting task becomes ready and continues the execution of the corresponding accept_statement (see 9.5.2).
- 10 For a call on an open entry of a protected object, the corresponding entry_body is executed (see 9.5.2) as part of the protected action.

11 If the accept_statement or entry_body completes other than by a requeue (see 9.5.4), return is made to the caller (after servicing the entry queues — see below); any necessary assigning back of formal to

actual parameters occurs, as for a subprogram call (see 6.4.1); such assignments take place outside of any protected action.

11.a **Ramification:** The return to the caller will generally not occur until the protected action completes, unless some other thread of control is given the job of completing the protected action and releasing the associated execution resource.

12 If the named entry is closed, the entry call is added to an *entry queue* (as part of the protected action, for a call on a protected entry), and the call remains queued until it is selected or cancelled; {*entry queue*} there is a separate (logical) entry queue for each entry of a given task or protected object [(including each entry of an entry family)].

13 {*service (an entry queue)*} {*select an entry call (from an entry queue)*} When a queued call is *selected*, it is removed from its entry queue. Selecting a queued call from a particular entry queue is called *servicing* the entry queue. An entry with queued calls can be serviced under the following circumstances:

14 • When the associated task reaches a corresponding accept_statement, or a selective_accept with a corresponding open accept_alternative;

15 • If after performing, as part of a protected action on the associated protected object, an operation on the object other than a call on a protected function, the entry is checked and found to be open.

16 {*select an entry call (from an entry queue)*} If there is at least one call on a queue corresponding to an open entry, then one such call is selected according to the *entry queuing policy* in effect (see below), and the corresponding accept_statement or entry_body is executed as above for an entry call that is selected immediately.

17 {*entry queuing policy*} The entry queuing policy controls selection among queued calls both for task and protected entry queues. {*default entry queuing policy*} {*entry queuing policy (default policy)*} The default entry queuing policy is to select calls on a given entry queue in order of arrival. If calls from two or more queues are simultaneously eligible for selection, the default entry queuing policy does not specify which queue is serviced first. Other entry queuing policies can be specified by pragmas (see D.4).

18 For a protected object, the above servicing of entry queues continues until there are no open entries with queued calls, at which point the protected action completes.

18.a **Discussion:** While servicing the entry queues of a protected object, no new calls can be added to any entry queue of the object, except due to an internal requeue (see 9.5.4). This is because the first step of a call on a protected entry is to start a new protected action, which implies acquiring (for exclusive read-write access) the execution resource associated with the protected object, which cannot be done while another protected action is already in progress.

19 {*blocked (during an entry call)* [partial]} For an entry call that is added to a queue, and that is not the triggering_statement of an asynchronous_select (see 9.7.4), the calling task is blocked until the call is cancelled, or the call is selected and a corresponding accept_statement or entry_body completes without requeuing. In addition, the calling task is blocked during a rendezvous.

19.a **Ramification:** For a call on a protected entry, the caller is not blocked if the call is selected immediately, unless a requeue causes the call to be queued.

20 {*cancellation (of an entry call)*} An attempt can be made to cancel an entry call upon an abort (see 9.8) and as part of certain forms of select_statement (see 9.7.2, 9.7.3, and 9.7.4). The cancellation does not take place until a point (if any) when the call is on some entry queue, and not protected from cancellation as part of a requeue (see 9.5.4); at such a point, the call is removed from the entry queue and the call completes due to the cancellation. The cancellation of a call on an entry of a protected object is a protected action[, and as such cannot take place while any other protected action is occurring on the protected object. Like any protected action, it includes servicing of the entry queues (in case some entry barrier depends on a Count attribute).]

20.a/2 **Implementation Note:** {*AI95-00114-01*} In the case of an attempted cancellation due to abort, this removal might have to be performed by the calling task itself if the ceiling priority of the protected object is lower than the priority of the task initiating the abort.

{*Tasking_Error (raised by failure of run-time check)*} A call on an entry of a task that has already completed its execution raises the exception Tasking_Error at the point of the call; similarly, this exception is raised at the point of the call if the called task completes its execution or becomes abnormal before accepting the call or completing the rendezvous (see 9.8). This applies equally to a simple entry call and to an entry call as part of a select_statement. 21

Implementation Permissions

An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action. If an entry_body completes without requeuing, then the corresponding calling task may be made ready without waiting for the entire protected action to complete. 22

Reason: These permissions are intended to allow flexibility for implementations on multiprocessors. On a monoprocessor, which thread of control executes the protected action is essentially invisible, since the thread is not abortable in any case, and the "current_task" function is not guaranteed to work during a protected action (see C.7.1). 22.a

When the entry of a protected object is checked to see whether it is open, the implementation need not reevaluate the condition of the corresponding entry_barrier if no variable or attribute referenced by the condition (directly or indirectly) has been altered by the execution (or cancellation) of a protected procedure or entry call on the object since the condition was last evaluated. 23

Ramification: Changes to variables referenced by an entry barrier that result from actions outside of a protected procedure or entry call on the protected object need not be "noticed." For example, if a global variable is referenced by an entry barrier, it should not be altered (except as part of a protected action on the object) any time after the barrier is first evaluated. In other words, globals can be used to "parameterize" a protected object, but they cannot reliably be used to control it after the first use of the protected object. 23.a

Implementation Note: Note that even if a global variable is volatile, the implementation need only reevaluate a barrier if the global is updated during a protected action on the protected object. This ensures that an entry-open bit-vector implementation approach is possible, where the bit-vector is computed at the end of a protected action, rather than upon each entry call. 23.b

An implementation may evaluate the conditions of all entry_barriers of a given protected object any time any entry of the object is checked to see if it is open. 24

Ramification: In other words, any side-effects of evaluating an entry barrier should be innocuous, since an entry barrier might be evaluated more or less often than is implied by the "official" dynamic semantics. 24.a

Implementation Note: It is anticipated that when the number of entries is known to be small, all barriers will be evaluated any time one of them needs to be, to produce an "entry-open bit-vector." The appropriate bit will be tested when the entry is called, and only if the bit is false will a check be made to see whether the bit-vector might need to be recomputed. This should allow an implementation to maximize the performance of a call on an open entry, which seems like the most important case. 24.b

In addition to the entry-open bit-vector, an "is-valid" bit is needed per object, which indicates whether the current bit-vector setting is valid. A "depends-on-Count-attribute" bit is needed per type. The "is-valid" bit is set to false (as are all the bits of the bit-vector) when the protected object is first created, as well as any time an exception is propagated from computing the bit-vector. Is-valid would also be set false any time the Count is changed and "depends-on-Count-attribute" is true for the type, or a protected procedure or entry returns indicating it might have updated a variable referenced in some barrier. 24.c

A single procedure can be compiled to evaluate all of the barriers, set the entry-open bit-vector accordingly, and set the is-valid bit to true. It could have a "when others" handler to set them all false, and call a routine to propagate Program_Error to all queued callers. 24.d

For protected types where the number of entries is not known to be small, it makes more sense to evaluate a barrier only when the corresponding entry is checked to see if it is open. It isn't worth saving the state of the entry between checks, because of the space that would be required. Furthermore, the entry queues probably want to take up space only when there is actually a caller on them, so rather than an array of all entry queues, a linked list of nonempty entry queues make the most sense in this case, with the first caller on each entry queue acting as the queue header. 24.e

When an attempt is made to cancel an entry call, the implementation need not make the attempt using the thread of control of the task (or interrupt) that initiated the cancellation; in particular, it may use the 25 thread of control of the caller itself to attempt the cancellation, even if this might allow the entry call to be selected in the interim.

25.a **Reason:** Because cancellation of a protected entry call is a protected action (which helps make the Count attribute of a protected entry meaningful), it might not be practical to attempt the cancellation from the thread of control that initiated the cancellation. For example, if the cancellation is due to the expiration of a delay, it is unlikely that the handler of the timer interrupt could perform the necessary protected action itself (due to being on the interrupt level). Similarly, if the cancellation is due to an abort, it is possible that the task initiating the abort has a priority higher than the ceiling priority of the protected object (for implementations that support ceiling priorities). Similar considerations could apply in a multiprocessor situation.

NOTES

26 27 If an exception is raised during the execution of an entry_body, it is propagated to the corresponding caller (see 11.4).

27 28 For a call on a protected entry, the entry is checked to see if it is open prior to queuing the call, and again thereafter if its Count attribute (see 9.9) is referenced in some entry barrier.

27.a **Ramification:** Given this, extra care is required if a reference to the Count attribute of an entry appears in the entry's own barrier.

27.b **Reason:** An entry is checked to see if it is open prior to queuing to maximize the performance of a call on an open entry.

28 29 In addition to simple entry calls, the language permits timed, conditional, and asynchronous entry calls (see 9.7.2, 9.7.3, and see 9.7.4).

28.a **Ramification:** A task can call its own entries, but the task will deadlock if the call is a simple entry call.

29 30 The condition of an entry_barrier is allowed to be evaluated by an implementation more often than strictly necessary, even if the evaluation might have side effects. On the other hand, an implementation need not reevaluate the condition if nothing it references was updated by an intervening protected action on the protected object, even if the condition references some global variable that might have been updated by an action performed from outside of a protected action.

Examples

30 *Examples of entry calls:*

31
```
Agent.Shut_Down;                      -- see 9.1
Parser.Next_Lexeme(E);                -- see 9.1
Pool(5).Read(Next_Char);              -- see 9.1
Controller.Request(Low)(Some_Item);   -- see 9.1
Flags(3).Seize;                       -- see 9.4
```

9.5.4 Requeue Statements

1 [A requeue_statement can be used to complete an accept_statement or entry_body, while redirecting the corresponding entry call to a new (or the same) entry queue. {*requeue*} Such a *requeue* can be performed with or without allowing an intermediate cancellation of the call, due to an abort or the expiration of a delay. {*preference control: See requeue*} {*broadcast signal: See requeue*}]

Syntax

2 requeue_statement ::= **requeue** *entry_*name [**with abort**];

Name Resolution Rules

3 {*target entry (of a requeue_statement)*} The *entry_*name of a requeue_statement shall resolve to denote an entry (the *target entry*) that either has no parameters, or that has a profile that is type conformant (see 6.3.1) with the profile of the innermost enclosing entry_body or accept_statement. {*type conformance (required)*}

Legality Rules

4 A requeue_statement shall be within a callable construct that is either an entry_body or an accept_statement, and this construct shall be the innermost enclosing body or callable construct.

5 If the target entry has parameters, then its profile shall be subtype conformant with the profile of the innermost enclosing callable construct. {*subtype conformance (required)*}

6 {*accessibility rule (requeue statement)* [partial]} In a requeue_statement of an accept_statement of some task unit, either the target object shall be a part of a formal parameter of the accept_statement, or the accessibility level of the target object shall not be equal to or statically deeper than any enclosing accept_statement of the task unit. In a requeue_statement of an entry_body of some protected unit, either the target object shall be a part of a formal parameter of the entry_body, or the accessibility level of the target object shall not be statically deeper than that of the entry_declaration.

6.a **Ramification:** In the entry_body case, the intent is that the target object can be global, or can be a component of the protected unit, but cannot be a local variable of the entry_body.

6.b **Reason:** These restrictions ensure that the target object of the requeue outlives the completion and finalization of the enclosing callable construct. They also prevent requeuing from a nested accept_statement on a parameter of an outer accept_statement, which could create some strange "long-distance" connections between an entry caller and its server.

6.c Note that in the strange case where a task_body is nested inside an accept_statement, it is permissible to requeue from an accept_statement of the inner task_body on parameters of the outer accept_statement. This is not a problem because all calls on the inner task have to complete before returning from the outer accept_statement, meaning no "dangling calls" will be created.

6.d **Implementation Note:** By disallowing certain requeues, we ensure that the normal terminate_alternative rules remain sensible, and that explicit clearing of the entry queues of a protected object during finalization is rarely necessary. In particular, such clearing of the entry queues is necessary only (ignoring premature Unchecked_Deallocation) for protected objects declared in a task_body (or created by an allocator for an access type declared in such a body) containing one or more requeue_statements. Protected objects declared in subprograms, or at the library level, will never need to have their entry queues explicitly cleared during finalization.

Dynamic Semantics

7 {*execution (requeue_statement)* [partial]} The execution of a requeue_statement proceeds by first evaluating the *entry_*name[, including the prefix identifying the target task or protected object and the expression identifying the entry within an entry family, if any]. The entry_body or accept_statement enclosing the requeue_statement is then completed[, finalized, and left (see 7.6.1)].

8 {*execution (requeue task entry)* [partial]} For the execution of a requeue on an entry of a target task, after leaving the enclosing callable construct, the named entry is checked to see if it is open and the requeued call is either selected immediately or queued, as for a normal entry call (see 9.5.3).

9 {*execution (requeue protected entry)* [partial]} For the execution of a requeue on an entry of a target protected object, after leaving the enclosing callable construct:

- 10 if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object — see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1);

10.a **Ramification:** Note that for an internal requeue, the call is queued without checking whether the target entry is open. This is because the entry queues will be serviced before the current protected action completes anyway, and considering the requeued call immediately might allow it to "jump" ahead of existing callers on the same queue.

- 11 if the requeue is an external requeue (that is, the target protected object is not implicitly the same as the current object — see 9.5), a protected action is started on the target object and proceeds as for a normal entry call (see 9.5.3).

12 If the new entry named in the requeue_statement has formal parameters, then during the execution of the accept_statement or entry_body corresponding to the new entry, the formal parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. [In any case, no parameters are specified in a requeue_statement; any parameter passing is implicit.]

13 {*requeue-with-abort*} If the requeue_statement includes the reserved words **with abort** (it is a *requeue-with-abort*), then:

14 - if the original entry call has been aborted (see 9.8), then the requeue acts as an abort completion point for the call, and the call is cancelled and no requeue is performed;

15 - if the original entry call was timed (or conditional), then the original expiration time is the expiration time for the requeued call.

16 If the reserved words **with abort** do not appear, then the call remains protected against cancellation while queued as the result of the requeue_statement.

16.a **Ramification:** This protection against cancellation lasts only until the call completes or a subsequent requeue-with-abort is performed on the call.

16.b **Reason:** We chose to protect a requeue, by default, against abort or cancellation. This seemed safer, since it is likely that extra steps need to be taken to allow for possible cancellation once the servicing of an entry call has begun. This also means that in the absence of **with abort** the usual Ada 83 behavior is preserved, namely that once an entry call is accepted, it cannot be cancelled until it completes.

NOTES

17 31 A requeue is permitted from a single entry to an entry of an entry family, or vice-versa. The entry index, if any, plays no part in the subtype conformance check between the profiles of the two entries; an entry index is part of the *entry*_name for an entry of a family. {*subtype conformance* [partial]}

Examples

18 *Examples of requeue statements:*

19
```
requeue Request(Medium) with abort;
                    -- requeue on a member of an entry family of the current task, see 9.1
```

20
```
requeue Flags(I).Seize;
                    -- requeue on an entry of an array component, see 9.4
```

Extensions to Ada 83

20.a {*extensions to Ada 83*} The requeue_statement is new.

9.6 Delay Statements, Duration, and Time

1 [{*expiration time* [partial]} A delay_statement is used to block further execution until a specified *expiration time* is reached. The expiration time can be specified either as a particular point in time (in a delay_until_statement), or in seconds from the current time (in a delay_relative_statement). The language-defined package Calendar provides definitions for a type Time and associated operations, including a function Clock that returns the current time. {*timing: See delay_statement*}]

Syntax

2 delay_statement ::= delay_until_statement | delay_relative_statement

3 delay_until_statement ::= **delay until** *delay*_expression;

4 delay_relative_statement ::= **delay** *delay*_expression;

Name Resolution Rules

5 {*expected type (delay_relative_statement expression)* [partial]} The expected type for the *delay*_expression in a delay_relative_statement is the predefined type Duration. {*expected type (delay_until_statement expression)* [partial]} The *delay*_expression in a delay_until_statement is expected to be of any nonlimited type.

Legality Rules

6 {*time type*} {*time base*} {*clock*} There can be multiple time bases, each with a corresponding clock, and a corresponding *time type*. The type of the *delay_*expression in a delay_until_statement shall be a time type — either the type Time defined in the language-defined package Calendar (see below), or some other implementation-defined time type (see D.8).

6.a **Implementation defined:** Any implementation-defined time types.

Static Semantics

7 [There is a predefined fixed point type named Duration, declared in the visible part of package Standard;] a value of type Duration is used to represent the length of an interval of time, expressed in seconds. [The type Duration is not specific to a particular time base, but can be used with any time base.]

8 A value of the type Time in package Calendar, or of some other implementation-defined time type, represents a time as reported by a corresponding clock.

9 The following language-defined library package exists:

10
```
package Ada.Calendar is
  type Time is private;
```

11/2
```
{AI95-00351-01}   subtype Year_Number  is Integer range 1901 .. 2399;
  subtype Month_Number is Integer range 1 .. 12;
  subtype Day_Number   is Integer range 1 .. 31;
  subtype Day_Duration is Duration range 0.0 .. 86_400.0;
```

11.a/2 **Reason:** {*AI95-00351-01*} A range of 500 years was chosen, as that only requires one extra bit for the year as compared to Ada 95. This was done to minimize disruptions with existing implementations. (One implementor reports that their time values represent nanoseconds, and this year range requires 63.77 bits to represent.)

12
```
  function Clock return Time;
```

13
```
  function Year   (Date : Time) return Year_Number;
  function Month  (Date : Time) return Month_Number;
  function Day    (Date : Time) return Day_Number;
  function Seconds(Date : Time) return Day_Duration;
```

14
```
  procedure Split (Date  : in Time;
                   Year    : out Year_Number;
                   Month   : out Month_Number;
                   Day     : out Day_Number;
                   Seconds : out Day_Duration);
```

15
```
  function Time_Of(Year  : Year_Number;
                   Month   : Month_Number;
                   Day     : Day_Number;
                   Seconds : Day_Duration := 0.0)
   return Time;
```

16
```
  function "+" (Left : Time;   Right : Duration) return Time;
  function "+" (Left : Duration; Right : Time) return Time;
  function "-" (Left : Time;   Right : Duration) return Time;
  function "-" (Left : Time;   Right : Time) return Duration;
```

17
```
  function "<" (Left, Right : Time) return Boolean;
  function "<="(Left, Right : Time) return Boolean;
  function ">" (Left, Right : Time) return Boolean;
  function ">="(Left, Right : Time) return Boolean;
```

18
```
  Time_Error : exception;
```

19
```
private
   ... -- not specified by the language
end Ada.Calendar;
```

Dynamic Semantics

20 {*execution (delay_statement)* [partial]} For the execution of a delay_statement, the *delay*_expression is first evaluated. {*expiration time (for a delay_until_statement)*} For a delay_until_statement, the expiration time for the delay is the value of the *delay*_expression, in the time base associated with the type of the expression. {*expiration time (for a delay_relative_statement)*} For a delay_relative_statement, the expiration time is defined as the current time, in the time base associated with relative delays, plus the value of the *delay*_expression converted to the type Duration, and then rounded up to the next clock tick. {*implicit subtype conversion (delay expression)* [partial]} The time base associated with relative delays is as defined in D.9, "Delay Accuracy" or is implementation defined.

20.a **Implementation defined:** The time base associated with relative delays.

20.b **Ramification:** Rounding up to the next clock tick means that the reading of the delay-relative clock when the delay expires should be no less than the current reading of the delay-relative clock plus the specified duration.

21 {*blocked (on a delay_statement)* [partial]} The task executing a delay_statement is blocked until the expiration time is reached, at which point it becomes ready again. If the expiration time has already passed, the task is not blocked.

21.a **Discussion:** For a delay_relative_statement, this case corresponds to when the value of the *delay*_expression is zero or negative.

21.b Even though the task is not blocked, it might be put back on the end of its ready queue. See D.2, "Priority Scheduling".

22 {*cancellation (of a delay_statement)*} If an attempt is made to *cancel* the delay_statement [(as part of an asynchronous_select or abort — see 9.7.4 and 9.8)], the _statement is cancelled if the expiration time has not yet passed, thereby completing the delay_statement.

22.a **Reason:** This is worded this way so that in an asynchronous_select where the triggering_statement is a delay_statement, an attempt to cancel the delay when the abortable_part completes is ignored if the expiration time has already passed, in which case the optional statements of the triggering_alternative are executed.

23 The time base associated with the type Time of package Calendar is implementation defined. The function Clock of package Calendar returns a value representing the current time for this time base. [The implementation-defined value of the named number System.Tick (see 13.7) is an approximation of the length of the real-time interval during which the value of Calendar.Clock remains constant.]

23.a **Implementation defined:** The time base of the type Calendar.Time.

24/2 {*AI95-00351-01*} The functions Year, Month, Day, and Seconds return the corresponding values for a given value of the type Time, as appropriate to an implementation-defined time zone; the procedure Split returns all four corresponding values. Conversely, the function Time_Of combines a year number, a month number, a day number, and a duration, into a value of type Time. The operators "+" and "–" for addition and subtraction of times and durations, and the relational operators for times, have the conventional meaning.

24.a/2 **Implementation defined:** The time zone used for package Calendar operations.

25 If Time_Of is called with a seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a seconds value of 0.0. The value returned by the function Seconds or through the Seconds parameter of the procedure Split is always less than 86_400.0.

26/1 {*8652/0030*} {*AI95-00113-01*} The exception Time_Error is raised by the function Time_Of if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "–" if the result is not representable in the type Time or Duration, as appropriate. This exception is also raised by the functions Year, Month, Day, and Seconds and the procedure Split if the year number of the given date is outside of the range of the subtype Year_Number.

26.a/1 **To be honest:** {*8652/0106*} {*AI95-00160-01*} By "proper date" above we mean that the given year has a month with the given day. For example, February 29th is a proper date only for a leap year. We do not mean to include the

Seconds in this notion; in particular, we do not mean to require implementations to check for the “missing hour” that occurs when Daylight Savings Time starts in the spring.

Reason: {*8652/0030*} {*AI95-00113-01*} {*AI95-00351-01*} We allow Year and Split to raise Time_Error because the arithmetic operators are allowed (but not required) to produce times that are outside the range of years from 1901 to 2399. This is similar to the way integer operators may return values outside the base range of their type so long as the value is mathematically correct. We allow the functions Month, Day and Seconds to raise Time_Error so that they can be implemented in terms of Split. 26.b/2

Implementation Requirements

The implementation of the type Duration shall allow representation of time intervals (both positive and negative) up to at least 86400 seconds (one day); Duration'Small shall not be greater than twenty milliseconds. The implementation of the type Time shall allow representation of all dates with year numbers in the range of Year_Number[; it may allow representation of other dates as well (both earlier and later).] 27

Implementation Permissions

An implementation may define additional time types (see D.8). 28

An implementation may raise Time_Error if the value of a *delay*_expression in a delay_until_statement of a select_statement represents a time more than 90 days past the current time. The actual limit, if any, is implementation-defined. 29

Implementation defined: Any limit on delay_until_statements of select_statements. 29.a

Implementation Note: This allows an implementation to implement select_statement timeouts using a representation that does not support the full range of a time type. In particular 90 days of seconds can be represented in 23 bits, allowing a signed 24-bit representation for the seconds part of a timeout. There is no similar restriction allowed for stand-alone delay_until_statements, as these can be implemented internally using a loop if necessary to accommodate a long delay. 29.b

Implementation Advice

Whenever possible in an implementation, the value of Duration'Small should be no greater than 100 microseconds. 30

Implementation Note: This can be satisfied using a 32-bit 2's complement representation with a *small* of 2.0**(–14) — that is, 61 microseconds — and a range of ± 2.0**17 — that is, 131_072.0. 30.a

Implementation Advice: The value of Duration'Small should be no greater than 100 microseconds. 30.b/2

The time base for delay_relative_statements should be monotonic; it need not be the same time base as used for Calendar.Clock. 31

Implementation Advice: The time base for delay_relative_statements should be monotonic. 31.a/2

NOTES

32 A delay_relative_statement with a negative value of the *delay*_expression is equivalent to one with a zero value. 32

33 A delay_statement may be executed by the environment task; consequently delay_statements may be executed as part of the elaboration of a library_item or the execution of the main subprogram. Such statements delay the environment task (see 10.2). 33

34 {*potentially blocking operation (delay_statement)* [partial]} {*blocking, potentially (delay_statement)* [partial]} A delay_statement is an abort completion point and a potentially blocking operation, even if the task is not actually blocked. 34

35 There is no necessary relationship between System.Tick (the resolution of the clock of package Calendar) and Duration'Small (the *small* of type Duration). 35

Ramification: The inaccuracy of the delay_statement has no relation to System.Tick. In particular, it is possible that the clock used for the delay_statement is less accurate than Calendar.Clock. 35.a

We considered making Tick a run-time-determined quantity, to allow for easier configurability. However, this would not be upward compatible, and the desired configurability can be achieved using functionality defined in Annex D, “Real-Time Systems”. 35.b

36 Additional requirements associated with delay_statements are given in D.9, “Delay Accuracy”. 36

Examples

37 *Example of a relative delay statement:*

38
```
delay 3.0;  -- delay 3.0 seconds
```

39 {*periodic task (example)*} {*periodic task: See delay_until_statement*} *Example of a periodic task:*

40
```
declare
   use Ada.Calendar;
   Next_Time : Time := Clock + Period;
                          -- Period is a global constant of type Duration
begin
   loop                   -- repeated every Period seconds
      delay until Next_Time;
      ... -- perform some actions
      Next_Time := Next_Time + Period;
   end loop;
end;
```

Inconsistencies With Ada 83

40.a {*inconsistencies with Ada 83*} For programs that raise Time_Error on "+" or "–" in Ada 83,the exception might be deferred until a call on Split or Year_Number, or might not be raised at all (if the offending time is never Split after being calculated). This should not affect typical programs, since they deal only with times corresponding to the relatively recent past or near future.

Extensions to Ada 83

40.b {*extensions to Ada 83*} The syntax rule for delay_statement is modified to allow delay_until_statements.

40.c/2 {*AI95-00351-01*} The type Time may represent dates with year numbers outside of Year_Number. Therefore, the operations "+" and "–" need only raise Time_Error if the result is not representable in Time (or Duration); also, Split or Year will now raise Time_Error if the year number is outside of Year_Number. This change is intended to simplify the implementation of "+" and "–" (allowing them to depend on overflow for detecting when to raise Time_Error) and to allow local time zone information to be considered at the time of Split rather than Clock (depending on the implementation approach). For example, in a POSIX environment, it is natural for the type Time to be based on GMT, and the results of procedure Split (and the functions Year, Month, Day, and Seconds) to depend on local time zone information. In other environments, it is more natural for the type Time to be based on the local time zone, with the results of Year, Month, Day, and Seconds being pure functions of their input.

40.d/2 *This paragraph was deleted.*{*AI95-00351-01*}

Inconsistencies With Ada 95

40.e/2 {*AI95-00351-01*} {*inconsistencies with Ada 95*} The upper bound of Year_Number has been changed to avoid a year 2100 problem. A program which expects years past 2099 to raise Constraint_Error will fail in Ada 2005. We don't expect there to be many programs which are depending on an exception to be raised. A program that uses Year_Number'Last as a magic number may also fail if values of Time are stored outside of the program. Note that the lower bound of Year_Number wasn't changed, because it is not unusual to use that value in a constant to represent an unknown time.

Wording Changes from Ada 95

40.f/2 {*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Clarified that Month, Day, and Seconds can raise Time_Error.

9.6.1 Formatting, Time Zones, and other operations for Time

Static Semantics

1/2 {*AI95-00351-01*} {*AI95-00427-01*} The following language-defined library packages exist:

2/2
```
package Ada.Calendar.Time_Zones is
```

3/2
```
   -- Time zone manipulation:
```

4/2
```
   type Time_Offset is range -28*60 .. 28*60;
```

Reason: We want to be able to specify the difference between any two arbitrary time zones. You might think that 1440 (24 hours) would be enough, but there are places (like Tonga, which is UTC+13hr) which are more than 12 hours than UTC. Combined with summer time (known as daylight saving time in some parts of the world) – which switches opposite in the northern and souther hemispheres – and even greater differences are possible. We know of cases of a 26 hours difference, so we err on the safe side by selecting 28 hours as the limit. 4.a/2

```
   Unknown_Zone_Error : exception;                                          5/2

   function UTC_Time_Offset (Date : Time := Clock) return Time_Offset;      6/2

end Ada.Calendar.Time_Zones;                                                7/2

                                                                            8/2
package Ada.Calendar.Arithmetic is

   -- Arithmetic on days:                                                   9/2

   type Day_Count is range                                                  10/2
     -366*(1+Year_Number'Last - Year_Number'First)
     ..
     366*(1+Year_Number'Last - Year_Number'First);

   subtype Leap_Seconds_Count is Integer range -2047 .. 2047;               11/2
```

Reason: The maximum number of leap seconds is likely to be much less than this, but we don't want to reach the limit too soon if the earth's behavior suddenly changes. We believe that the maximum number is 1612, based on the current rules, but that number is too weird to use here. 11.a/2

```
   procedure Difference (Left, Right : in Time;                             12/2
                         Days : out Day_Count;
                         Seconds : out Duration;
                         Leap_Seconds : out Leap_Seconds_Count);

   function "+" (Left : Time; Right : Day_Count) return Time;               13/2
   function "+" (Left : Day_Count; Right : Time) return Time;
   function "-" (Left : Time; Right : Day_Count) return Time;
   function "-" (Left, Right : Time) return Day_Count;

end Ada.Calendar.Arithmetic;                                                14/2

                                                                            15/2
with Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting is

   -- Day of the week:                                                      16/2

   type Day_Name is (Monday, Tuesday, Wednesday, Thursday,                  17/2
       Friday, Saturday, Sunday);

   function Day_of_Week (Date : Time) return Day_Name;                      18/2

   -- Hours:Minutes:Seconds access:                                         19/2

   subtype Hour_Number         is Natural range 0 .. 23;                    20/2
   subtype Minute_Number       is Natural range 0 .. 59;
   subtype Second_Number       is Natural range 0 .. 59;
   subtype Second_Duration     is Day_Duration range 0.0 .. 1.0;

   function Year       (Date : Time;                                        21/2
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Year_Number;

   function Month      (Date : Time;                                        22/2
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Month_Number;

   function Day        (Date : Time;                                        23/2
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Day_Number;

   function Hour       (Date : Time;                                        24/2
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Hour_Number;

   function Minute     (Date : Time;                                        25/2
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Minute_Number;
```

26/2
```
      function Second     (Date : Time)
                              return Second_Number;
```

27/2
```
      function Sub_Second (Date : Time)
                              return Second_Duration;
```

28/2
```
      function Seconds_Of (Hour   :  Hour_Number;
                           Minute : Minute_Number;
                           Second : Second_Number := 0;
                           Sub_Second : Second_Duration := 0.0)
          return Day_Duration;
```

29/2
```
      procedure Split (Seconds    : in Day_Duration;
                       Hour       : out Hour_Number;
                       Minute     : out Minute_Number;
                       Second     : out Second_Number;
                       Sub_Second : out Second_Duration);
```

30/2
```
      function Time_Of (Year       : Year_Number;
                        Month      : Month_Number;
                        Day        : Day_Number;
                        Hour       : Hour_Number;
                        Minute     : Minute_Number;
                        Second     : Second_Number;
                        Sub_Second : Second_Duration := 0.0;
                        Leap_Second: Boolean := False;
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                                return Time;
```

31/2
```
      function Time_Of (Year       : Year_Number;
                        Month      : Month_Number;
                        Day        : Day_Number;
                        Seconds    : Day_Duration := 0.0;
                        Leap_Second: Boolean := False;
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                                return Time;
```

32/2
```
      procedure Split (Date       : in Time;
                       Year       : out Year_Number;
                       Month      : out Month_Number;
                       Day        : out Day_Number;
                       Hour       : out Hour_Number;
                       Minute     : out Minute_Number;
                       Second     : out Second_Number;
                       Sub_Second : out Second_Duration;
                       Time_Zone  : in Time_Zones.Time_Offset := 0);
```

33/2
```
      procedure Split (Date       : in Time;
                       Year       : out Year_Number;
                       Month      : out Month_Number;
                       Day        : out Day_Number;
                       Hour       : out Hour_Number;
                       Minute     : out Minute_Number;
                       Second     : out Second_Number;
                       Sub_Second : out Second_Duration;
                       Leap_Second: out Boolean;
                       Time_Zone  : in Time_Zones.Time_Offset := 0);
```

34/2
```
      procedure Split (Date       : in Time;
                       Year       : out Year_Number;
                       Month      : out Month_Number;
                       Day        : out Day_Number;
                       Seconds    : out Day_Duration;
                       Leap_Second: out Boolean;
                       Time_Zone  : in Time_Zones.Time_Offset := 0);
```

35/2
```
      -- Simple image and value:
      function Image (Date : Time;
                      Include_Time_Fraction : Boolean := False;
                      Time_Zone  : Time_Zones.Time_Offset := 0) return String;
```

36/2
```
      function Value (Date : String;
                      Time_Zone  : Time_Zones.Time_Offset := 0) return Time;
```

```
   function Image (Elapsed_Time : Duration;                                37/2
                   Include_Time_Fraction : Boolean := False) return String;
```

```
   function Value (Elapsed_Time : String) return Duration;                 38/2
```

```
end Ada.Calendar.Formatting;                                               39/2
```

{*AI95-00351-01*} Type Time_Offset represents the number of minutes difference between the implementation-defined time zone used by Calendar and another time zone. 40/2

```
function UTC_Time_Offset (Date : Time := Clock) return Time_Offset;         41/2
```

{*AI95-00351-01*} Returns, as a number of minutes, the difference between the implementation-defined time zone of Calendar, and UTC time, at the time Date. If the time zone of the Calendar implementation is unknown, then Unknown_Zone_Error is raised. 42/2

Discussion: The Date parameter is needed to take into account time differences caused by daylight-savings time and other time changes. This parameter is measured in the time zone of Calendar, if any, not necessarily the UTC time zone. 42.a/2

Other time zones can be supported with a child package. We don't define one because of the lack of agreement on the definition of a time zone. 42.b/2

The accuracy of this routine is not specified; the intent is that the facilities of the underlying target operating system are used to implement it. 42.c/2

```
procedure Difference (Left, Right : in Time;                                43/2
                      Days : out Day_Count;
                      Seconds : out Duration;
                      Leap_Seconds : out Leap_Seconds_Count);
```

{*AI95-00351-01*} {*AI95-00427-01*} Returns the difference between Left and Right. Days is the number of days of difference, Seconds is the remainder seconds of difference excluding leap seconds, and Leap_Seconds is the number of leap seconds. If Left < Right, then Seconds <= 0.0, Days <= 0, and Leap_Seconds <= 0. Otherwise, all values are nonnegative. The absolute value of Seconds is always less than 86_400.0. For the returned values, if Days = 0, then Seconds + Duration(Leap_Seconds) = Calendar."–" (Left, Right). 44/2

Discussion: Leap_Seconds, if any, are not included in Seconds. However, Leap_Seconds should be included in calculations using the operators defined in Calendar, as is specified for "–" above. 44.a/2

```
function "+" (Left : Time; Right : Day_Count) return Time;                  45/2
function "+" (Left : Day_Count; Right : Time) return Time;
```

{*AI95-00351-01*} Adds a number of days to a time value. Time_Error is raised if the result is not representable as a value of type Time. 46/2

```
function "-" (Left : Time; Right : Day_Count) return Time;                  47/2
```

{*AI95-00351-01*} Subtracts a number of days from a time value. Time_Error is raised if the result is not representable as a value of type Time. 48/2

```
function "-" (Left, Right : Time) return Day_Count;                         49/2
```

{*AI95-00351-01*} Subtracts two time values, and returns the number of days between them. This is the same value that Difference would return in Days. 50/2

```
function Day_of_Week (Date : Time) return Day_Name;                         51/2
```

{*AI95-00351-01*} Returns the day of the week for Time. This is based on the Year, Month, and Day values of Time. 52/2

```
function Year       (Date : Time;                                           53/2
                      Time_Zone  : Time_Zones.Time_Offset := 0)
                         return Year_Number;
```

{*AI95-00427-01*} Returns the year for Date, as appropriate for the specified time zone offset. 54/2

55/2
```
function Month      (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Month_Number;
```

56/2 {*AI95-00427-01*} Returns the month for Date, as appropriate for the specified time zone offset.

57/2
```
function Day        (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Day_Number;
```

58/2 {*AI95-00427-01*} Returns the day number for Date, as appropriate for the specified time zone offset.

59/2
```
function Hour       (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Hour_Number;
```

60/2 {*AI95-00351-01*} Returns the hour for Date, as appropriate for the specified time zone offset.

61/2
```
function Minute     (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Minute_Number;
```

62/2 {*AI95-00351-01*} Returns the minute within the hour for Date, as appropriate for the specified time zone offset.

63/2
```
function Second     (Date : Time)
                        return Second_Number;
```

64/2 {*AI95-00351-01*} {*AI95-00427-01*} Returns the second within the hour and minute for Date.

65/2
```
function Sub_Second (Date : Time)
                        return Second_Duration;
```

66/2 {*AI95-00351-01*} {*AI95-00427-01*} Returns the fraction of second for Date (this has the same accuracy as Day_Duration). The value returned is always less than 1.0.

67/2
```
function Seconds_Of (Hour   : Hour_Number;
                     Minute : Minute_Number;
                     Second : Second_Number := 0;
                     Sub_Second : Second_Duration := 0.0)
    return Day_Duration;
```

68/2 {*AI95-00351-01*} {*AI95-00427-01*} Returns a Day_Duration value for the combination of the given Hour, Minute, Second, and Sub_Second. This value can be used in Calendar.Time_Of as well as the argument to Calendar."+" and Calendar."–". If Seconds_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Seconds_Of for the next second with a Sub_Second value of 0.0.

69/2
```
procedure Split (Seconds    : in Day_Duration;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration);
```

70/2 {*AI95-00351-01*} {*AI95-00427-01*} Splits Seconds into Hour, Minute, Second and Sub_Second in such a way that the resulting values all belong to their respective subtypes. The value returned in the Sub_Second parameter is always less than 1.0.

70.a/2 **Ramification:** There is only one way to do the split which meets all of the requirements.

71/2
```
function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Hour       : Hour_Number;
                  Minute     : Minute_Number;
                  Second     : Second_Number;
                  Sub_Second : Second_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                          return Time;
```

72/2 {*AI95-00351-01*} {*AI95-00427-01*} If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Time_Of for the next second with a Sub_Second value of 0.0.

72.a/2 **Discussion:** Time_Error should be raised if Leap_Second is True, and the date and time values do not represent the second before a leap second. A leap second always occurs at midnight UTC, and is 23:59:60 UTC in ISO notation. So, if the time zone is UTC and Leap_Second is True, if any of Hour /= 23, Minute /= 59, or Second /= 59, then Time_Error should be raised. However, we do not say that, because other time zones will have different values where a leap second is allowed.

73/2
```
function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Seconds    : Day_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                          return Time;
```

74/2 {*AI95-00351-01*} {*AI95-00427-01*} If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a Seconds value of 0.0.

75/2
```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

76/2 {*AI95-00351-01*} {*AI95-00427-01*} If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Sub_Second parameter is always less than 1.0.

77/2
```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

78/2 {*AI95-00351-01*} {*AI95-00427-01*} Splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset. The value returned in the Sub_Second parameter is always less than 1.0.

79/2
```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Seconds    : out Day_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

80/2 {*AI95-00351-01*} {*AI95-00427-01*} If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Seconds), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Seconds parameter is always less than 86_400.0.

81/2
```
function Image (Date : Time;
                Include_Time_Fraction : Boolean := False;
                Time_Zone  : Time_Zones.Time_Offset := 0) return String;
```

82/2 {*AI95-00351-01*} Returns a string form of the Date relative to the given Time_Zone. The format is "Year-Month-Day Hour:Minute:Second", where the Year is a 4-digit value, and all others are 2-digit values, of the functions defined in Calendar and Calendar.Formatting, including a leading zero, if needed. The separators between the values are a minus, another minus, a colon, and a single space between the Day and Hour. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value.

82.a/2 **Discussion:** The Image provides a string in ISO 8601 format, the international standard time format. Alternative representations allowed in ISO 8601 are not supported here.

82.b/2 ISO 8601 allows 24:00:00 for midnight; and a seconds value of 60 for leap seconds. These are not allowed here (the routines mentioned above cannot produce those results).

82.c/2 **Ramification:** The fractional part is truncated, not rounded. It would be quite hard to define the result with proper rounding, as it can change all of the values of the image. Values can be rounded up by adding an appropriate constant (0.5 if Include_Time_Fraction is False, 0.005 otherwise) to the time before taking the image.

83/2
```
function Value (Date : String;
                Time_Zone  : Time_Zones.Time_Offset := 0) return Time;
```

84/2 {*AI95-00351-01*} Returns a Time value for the image given as Date, relative to the given time zone. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Time value.

84.a/2 **Discussion:** The intent is that the implementation enforce the same range rules on the string as the appropriate function Time_Of, except for the hour, so “cannot interpret the given string as a Time value” happens when one of the values is out of the required range. For example, "2005-08-31 24:0:0" should raise Constraint_Error (the hour is out of range).

```
function Image (Elapsed_Time : Duration;
                Include_Time_Fraction : Boolean := False) return String;
```
85/2

86/2 {*AI95-00351-01*} Returns a string form of the Elapsed_Time. The format is "Hour:Minute:Second", where all values are 2-digit values, including a leading zero, if needed. The separators between the values are colons. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value. If Elapsed_Time < 0.0, the result is Image (**abs** Elapsed_Time, Include_Time_Fraction) prefixed with a minus sign. If **abs** Elapsed_Time represents 100 hours or more, the result is implementation-defined.

86.a/2 **Implementation defined:** The result of Calendar.Formating.Image if its argument represents more than 100 hours.

86.b/2 **Implementation Note:** This cannot be implemented (directly) by calling Calendar.Formatting.Split, since it may be out of the range of Day_Duration, and thus the number of hours may be out of the range of Hour_Number.

86.c If a Duration value can represent more then 100 hours, the implementation will need to define a format for the return of Image.

```
function Value (Elapsed_Time : String) return Duration;
```
87/2

88/2 {*AI95-00351-01*} Returns a Duration value for the image given as Elapsed_Time. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Duration value.

88.a/2 **Discussion:** The intent is that the implementation enforce the same range rules on the string as the appropriate function Time_Of, except for the hour, so “cannot interpret the given string as a Time value” happens when one of the values is out of the required range. For example, "10:23:60" should raise Constraint_Error (the seconds value is out of range).

Implementation Advice

89/2 {*AI95-00351-01*} An implementation should support leap seconds if the target system supports them. If leap seconds are not supported, Difference should return zero for Leap_Seconds, Split should return False for Leap_Second, and Time_Of should raise Time_Error if Leap_Second is True.

89.a/2 **Implementation Advice:** Leap seconds should be supported if the target system supports them. Otherwise, operations in Calendar.Formatting should return results consistent with no leap seconds.

89.b/2 **Discussion:** An implementation can always support leap seconds when the target system does not; indeed, this isn't particularly hard (all that is required is a table of when leap seconds were inserted). As such, leap second support isn't “impossible or impractical” in the sense of 1.1.3. However, for some purposes, it may be important to follow the target system's lack of leap second support (if the target is a GPS satellite, which does not use leap seconds, leap second support would be a handicap to work around). Thus, this Implementation Advice should be read as giving permission to not support leap seconds on target systems that don't support leap seconds. Implementers should use the needs of their customers to determine whether or not support leap seconds on such targets.

NOTES

90/2 37 {*AI95-00351-01*} The implementation-defined time zone of package Calendar may, but need not, be the local time zone. UTC_Time_Offset always returns the difference relative to the implementation-defined time zone of package Calendar. If UTC_Time_Offset does not raise Unknown_Zone_Error, UTC time can be safely calculated (within the accuracy of the underlying time-base).

90.a/2 **Discussion:** {*AI95-00351-01*} The time in the time zone known as Greenwich Mean Time (GMT) is generally very close to UTC time; for most purposes they can be treated the same. GMT is the time based on the rotation of the Earth; UTC is the time based on atomic clocks, with leap seconds periodically inserted to realign with GMT (because most human activities depend on the rotation of the Earth). At any point in time, there will be a sub-second difference between GMT and UTC.

91/2 38 {*AI95-00351-01*} Calling Split on the results of subtracting Duration(UTC_Time_Offset*60) from Clock provides the components (hours, minutes, and so on) of the UTC time. In the United States, for example, UTC_Time_Offset will generally be negative.

91.a/2 **Discussion:** This is an illustration to help specify the value of UTC_Time_Offset. A user should pass UTC_Time_Offset as the Time_Zone parameter of Split, rather than trying to make the above calculation.

Extensions to Ada 95

91.b/2 {*AI95-00351-01*} {*AI95-00428-01*} {*extensions to Ada 95*} Packages Calendar.Time_Zones, Calendar.Arithmetic, and Calendar.Formatting are new.

9.7 Select Statements

1 [There are four forms of the select_statement. One form provides a selective wait for one or more select_alternatives. Two provide timed and conditional entry calls. The fourth provides asynchronous transfer of control.]

Syntax

2
```
select_statement ::=
   selective_accept
  | timed_entry_call
  | conditional_entry_call
  | asynchronous_select
```

Examples

3 *Example of a select statement:*

4
```
select
   accept Driver_Awake_Signal;
or
   delay 30.0*Seconds;
   Stop_The_Train;
end select;
```

Extensions to Ada 83

4.a {*extensions to Ada 83*} Asynchronous_select is new.

9.7.1 Selective Accept

1 [This form of the select_statement allows a combination of waiting for, and selecting from, one or more alternatives. The selection may depend on conditions associated with each alternative of the selective_accept. {*time-out: See selective_accept*}]

Syntax

2
```
selective_accept ::=
  select
   [guard]
     select_alternative
{ or
   [guard]
     select_alternative }
[ else
   sequence_of_statements ]
  end select;
```

3
```
guard ::= when condition =>
```

4
```
select_alternative ::=
   accept_alternative
  | delay_alternative
  | terminate_alternative
```

accept_alternative ::=
  accept_statement [sequence_of_statements] 5

delay_alternative ::=
  delay_statement [sequence_of_statements] 6

terminate_alternative ::= **terminate**; 7

A selective_accept shall contain at least one accept_alternative. In addition, it can contain: 8

- a terminate_alternative (only one); or 9
- one or more delay_alternatives; or 10
- {*else part (of a selective_accept)*} an *else part* (the reserved word **else** followed by a sequence_of_statements). 11

These three possibilities are mutually exclusive. 12

Legality Rules

13 If a selective_accept contains more than one delay_alternative, then all shall be delay_relative_-statements, or all shall be delay_until_statements for the same time type.

13.a **Reason:** This simplifies the implementation and the description of the semantics.

Dynamic Semantics

14 {*open alternative*} A select_alternative is said to be *open* if it is not immediately preceded by a guard, or if the condition of its guard evaluates to True. It is said to be *closed* otherwise.

15 {*execution (selective_accept)* [partial]} For the execution of a selective_accept, any guard conditions are evaluated; open alternatives are thus determined. For an open delay_alternative, the *delay*_expression is also evaluated. Similarly, for an open accept_alternative for an entry of a family, the entry_index is also evaluated. These evaluations are performed in an arbitrary order, except that a *delay*_expression or entry_index is not evaluated until after evaluating the corresponding condition, if any. Selection and execution of one open alternative, or of the else part, then completes the execution of the selective_accept; the rules for this selection are described below.

16 Open accept_alternatives are first considered. Selection of one such alternative takes place immediately if the corresponding entry already has queued calls. If several alternatives can thus be selected, one of them is selected according to the entry queuing policy in effect (see 9.5.3 and D.4). When such an alternative is selected, the selected call is removed from its entry queue and the handled_sequence_of_-statements (if any) of the corresponding accept_statement is executed; after the rendezvous completes any subsequent sequence_of_statements of the alternative is executed. {*blocked (execution of a selective_accept)* [partial]} If no selection is immediately possible (in the above sense) and there is no else part, the task blocks until an open alternative can be selected.

17 Selection of the other forms of alternative or of an else part is performed as follows:

- 18 An open delay_alternative is selected when its expiration time is reached if no accept_-alternative or other delay_alternative can be selected prior to the expiration time. If several delay_alternatives have this same expiration time, one of them is selected according to the queuing policy in effect (see D.4); the default queuing policy chooses arbitrarily among the delay_alternatives whose expiration time has passed.
- 19 The else part is selected and its sequence_of_statements is executed if no accept_alternative can immediately be selected; in particular, if all alternatives are closed.
- 20 An open terminate_alternative is selected if the conditions stated at the end of clause 9.3 are satisfied.

20.a **Ramification:** In the absence of a requeue_statement, the conditions stated are such that a terminate_alternative cannot be selected while there is a queued entry call for any entry of the task. In the presence of requeues from a task to one of its subtasks, it is possible that when a terminate_alternative of the subtask is selected, requeued calls (for closed entries only) might still be queued on some entry of the subtask. Tasking_Error will be propagated to such callers, as is usual when a task completes while queued callers remain.

21 {*Program_Error (raised by failure of run-time check)*} The exception Program_Error is raised if all alternatives are closed and there is no else part.

NOTES

22 39 A selective_accept is allowed to have several open delay_alternatives. A selective_accept is allowed to have several open accept_alternatives for the same entry.

Examples

23 *Example of a task body with a selective accept:*

24
```
task body Server is
   Current_Work_Item : Work_Item;
begin
   loop
      select
         accept Next_Work_Item(WI : in Work_Item) do
            Current_Work_Item := WI;
          end;
          Process_Work_Item(Current_Work_Item);
      or
         accept Shut_Down;
         exit;         -- Premature shut down requested
      or
         terminate;    -- Normal shutdown at end of scope
      end select;
   end loop;
end Server;
```

Wording Changes from Ada 83

24.a The name of selective_wait was changed to selective_accept to better describe what is being waited for. We kept select_alternative as is, because selective_accept_alternative was too easily confused with accept_alternative.

9.7.2 Timed Entry Calls

1/2 {*AI95-00345-01*} [A timed_entry_call issues an entry call that is cancelled if the call (or a requeue-with-abort of the call) is not selected before the expiration time is reached. A procedure call may appear rather than an entry call for cases where the procedure might be implemented by an entry. {*time-out: See timed_entry_call*}]

Syntax

2 timed_entry_call ::=
  select
   entry_call_alternative
  or
   delay_alternative
  end select;

3/2 {*AI95-00345-01*} entry_call_alternative ::=
  procedure_or_entry_call [sequence_of_statements]

3.1/2 {*AI95-00345-01*} procedure_or_entry_call ::=
  procedure_call_statement | entry_call_statement

Legality Rules

{*AI95-00345-01*} If a procedure_call_statement is used for a procedure_or_entry_call, the *procedure*_name or *procedure*_prefix of the procedure_call_statement shall statically denote an entry renamed as a procedure or (a view of) a primitive subprogram of a limited interface whose first parameter is a controlling parameter (see 3.9.2). 3.2/2

> **Reason:** This would be a confusing way to call a procedure, so we only allow it when it is possible that the procedure is actually an entry. We could have allowed formal subprograms here, but we didn't because we'd have to allow all formal subprograms, and it would increase the difficulty of generic code sharing. 3.a/2

> We say “statically denotes” because an access-to-subprogram cannot be primitive, and we don't have anything like access-to-entry. So only names of entries or procedures are possible. 3.b/2

Static Semantics

{*AI95-00345-01*} If a procedure_call_statement is used for a procedure_or_entry_call, and the procedure is implemented by an entry, then the *procedure*_name, or *procedure*_prefix and possibly the first parameter of the procedure_call_statement, determine the target object of the call and the entry to be called. 3.3/2

> **Discussion:** The above says “possibly the first parameter”, because Ada allows entries to be renamed and passed as formal subprograms. In those cases, the task or protected object is implicit in the name of the routine; otherwise the object is an explicit parameter to the call. 3.c/2

Dynamic Semantics

{*AI95-00345-01*} {*execution (timed_entry_call)* [partial]} For the execution of a timed_entry_call, the *entry*_name, *procedure*_name, or *procedure*_prefix, and any actual parameters are evaluated, as for a simple entry call (see 9.5.3) or procedure call (see 6.4). The expiration time (see 9.6) for the call is determined by evaluating the *delay*_expression of the delay_alternative. If the call is an entry call or a call on a procedure implemented by an entry, the entry call is then issued. Otherwise, the call proceeds as described in 6.4 for a procedure call, followed by the sequence_of_statements of the entry_call_-alternative; the sequence_of_statements of the delay_alternative is ignored. 4/2

If the call is queued (including due to a requeue-with-abort), and not selected before the expiration time is reached, an attempt to cancel the call is made. If the call completes due to the cancellation, the optional sequence_of_statements of the delay_alternative is executed; if the entry call completes normally, the optional sequence_of_statements of the entry_call_alternative is executed. 5

> *This paragraph was deleted.*{*AI95-00345-01*} 5.a/2

Examples

Example of a timed entry call: 6

7
```
select
   Controller.Request(Medium)(Some_Item);
or
   delay 45.0;
   --  controller too busy, try something else
end select;
```

Wording Changes from Ada 83

> This clause comes before the one for Conditional Entry Calls, so we can define conditional entry calls in terms of timed entry calls. 7.a

Incompatibilities With Ada 95

> {*AI95-00345-01*} {*incompatibilities with Ada 95*} A procedure can be used as the in a timed or conditional entry call, if the procedure might actually be an entry. Since the fact that something is an entry could be used in resolving these calls in Ada 95, it is possible for timed or conditional entry calls that resolved in Ada 95 to be ambiguous in Ada 2005. That could happen if both an entry and procedure with the same name and profile exist, which should be rare. 7.b/2

9.7.3 Conditional Entry Calls

1/2 {*AI95-00345-01*} [A conditional_entry_call issues an entry call that is then cancelled if it is not selected immediately (or if a requeue-with-abort of the call is not selected immediately). A procedure call may appear rather than an entry call for cases where the procedure might be implemented by an entry.]

1.a **To be honest:** In the case of an entry call on a protected object, it is OK if the entry is closed at the start of the corresponding protected action, so long as it opens and the call is selected before the end of that protected action (due to changes in the Count attribute).

Syntax

2 conditional_entry_call ::=
  select
   entry_call_alternative
  else
   sequence_of_statements
  end select;

Dynamic Semantics

3 {*execution (conditional_entry_call)* [partial]} The execution of a conditional_entry_call is defined to be equivalent to the execution of a timed_entry_call with a delay_alternative specifying an immediate expiration time and the same sequence_of_statements as given after the reserved word **else**.

NOTES

4 40 A conditional_entry_call may briefly increase the Count attribute of the entry, even if the conditional call is not selected.

Examples

5 *Example of a conditional entry call:*

6
```
procedure Spin(R : in Resource) is
begin
   loop
      select
         R.Seize;
         return;
      else
         null;  -- busy waiting
      end select;
   end loop;
end;
```

Wording Changes from Ada 83

6.a This clause comes after the one for Timed Entry Calls, so we can define conditional entry calls in terms of timed entry calls. We do that so that an "expiration time" is defined for both, thereby simplifying the definition of what happens on a requeue-with-abort.

9.7.4 Asynchronous Transfer of Control

1 [An asynchronous select_statement provides asynchronous transfer of control upon completion of an entry call or the expiration of a delay.]

Syntax

```
asynchronous_select ::=
  select
   triggering_alternative
  then abort
   abortable_part
  end select;
```
2

triggering_alternative ::= triggering_statement [sequence_of_statements] 3

{*AI95-00345-01*} triggering_statement ::= procedure_or_entry_call | delay_statement 4/2

abortable_part ::= sequence_of_statements 5

Dynamic Semantics

6/2 {*AI95-00345-01*} {*execution (asynchronous_select with an entry call trigger)* [partial]} {*execution (asynchronous_select with a procedure call trigger)* [partial]} For the execution of an asynchronous_select whose triggering_statement is a procedure_or_entry_call, the *entry*_name, *procedure*_name, or *procedure*_prefix, and actual parameters are evaluated as for a simple entry call (see 9.5.3) or procedure call (see 6.4). If the call is an entry call or a call on a procedure implemented by an entry, the entry call is issued. If the entry call is queued (or requeued-with-abort), then the abortable_part is executed. [If the entry call is selected immediately, and never requeued-with-abort, then the abortable_part is never started.] If the call is on a procedure that is not implemented by an entry, the call proceeds as described in 6.4, followed by the sequence_of_statements of the triggering_alternative[; the abortable_part is never started].

7 {*execution (asynchronous_select with a delay_statement trigger)* [partial]} For the execution of an asynchronous_select whose triggering_statement is a delay_statement, the *delay*_expression is evaluated and the expiration time is determined, as for a normal delay_statement. If the expiration time has not already passed, the abortable_part is executed.

8 If the abortable_part completes and is left prior to completion of the triggering_statement, an attempt to cancel the triggering_statement is made. If the attempt to cancel succeeds (see 9.5.3 and 9.6), the asynchronous_select is complete.

9 If the triggering_statement completes other than due to cancellation, the abortable_part is aborted (if started but not yet completed — see 9.8). If the triggering_statement completes normally, the optional sequence_of_statements of the triggering_alternative is executed after the abortable_part is left.

9.a **Discussion:** We currently don't specify when the by-copy [**in**] **out** parameters are assigned back into the actuals. We considered requiring that to happen after the abortable_part is left. However, that doesn't seem useful enough to justify possibly overspecifying the implementation approach, since some of the parameters are passed by reference anyway.

9.b In an earlier description, we required that the sequence_of_statements of the triggering_alternative execute after aborting the abortable_part, but before waiting for it to complete and finalize, to provide more rapid response to the triggering event in case the finalization was unbounded. However, various reviewers felt that this created unnecessary complexity in the description, and a potential for undesirable concurrency (and nondeterminism) within a single task. We have now reverted to simpler, more deterministic semantics, but anticipate that further discussion of this issue might be appropriate during subsequent reviews. One possibility is to leave this area implementation defined, so as to encourage experimentation. The user would then have to assume the worst about what kinds of actions are appropriate for the sequence_of_statements of the triggering_alternative to achieve portability.

Examples

10 {*signal handling (example)*} {*interrupt (example using asynchronous_select)*} {*terminal interrupt (example)*} *Example of a main command loop for a command interpreter:*

11
```
loop
    select
        Terminal.Wait_For_Interrupt;
        Put_Line("Interrupted");
    then abort
        -- This will be abandoned upon terminal interrupt
        Put_Line("-> ");
        Get_Line(Command, Last);
        Process_Command(Command(1..Last));
    end select;
end loop;
```

12 *Example of a time-limited calculation:* {*time-out: See asynchronous_select*} {*time-out (example)*} {*time limit (example)*} {*interrupt (example using asynchronous_select)*} {*timer interrupt (example)*}

13
```
select
   delay 5.0;
   Put_Line("Calculation does not converge");
then abort
   -- This calculation should finish in 5.0 seconds;
   --  if not, it is assumed to diverge.
   Horribly_Complicated_Recursive_Function(X, Y);
end select;
```

Extensions to Ada 83

13.a {*extensions to Ada 83*} Asynchronous_select is new.

Extensions to Ada 95

13.b/2 {*AI95-00345-01*} {*extensions to Ada 95*} A procedure can be used as the triggering_statement of an asynchronous_select, if the procedure might actually be an entry

9.8 Abort of a Task - Abort of a Sequence of Statements

1 [An abort_statement causes one or more tasks to become abnormal, thus preventing any further interaction with such tasks. The completion of the triggering_statement of an asynchronous_select causes a sequence_of_statements to be aborted.]

Syntax

2 abort_statement ::= **abort** *task*_name {, *task*_name};

Name Resolution Rules

3 {*expected type (abort_statement task_name)* [partial]} Each *task*_name is expected to be of any task type[; they need not all be of the same task type.]

Dynamic Semantics

4 {*execution (abort_statement)* [partial]} For the execution of an abort_statement, the given *task*_names are evaluated in an arbitrary order. {*abort (of a task)*} {*abnormal task*} {*task state (abnormal)* [partial]} Each named task is then *aborted*, which consists of making the task *abnormal* and aborting the execution of the corresponding task_body, unless it is already completed.

4.a/2 **Ramification:** {*AI95-00114-01*} Note that aborting those tasks is not defined to be an abort-deferred operation. Therefore, if one of the named tasks is the task executing the abort_statement, or if the task executing the abort_statement depends on one of the named tasks, then it is possible for the execution of the abort_statement to be aborted, thus leaving some of the tasks unaborted. This allows the implementation to use either a sequence of calls to an "abort task" run-time system primitive, or a single call to an "abort list of tasks" run-time system primitive.

5 {*execution (aborting the execution of a construct)* [partial]} {*abort (of the execution of a construct)*} When the execution of a construct is *aborted* (including that of a task_body or of a sequence_of_statements), the execution of every construct included within the aborted execution is also aborted, except for executions included within the execution of an *abort-deferred* operation; the execution of an abort-deferred operation continues to completion without being affected by the abort; {*abort-deferred operation*} the following are the abort-deferred operations:

- 6 a protected action;
- 7 waiting for an entry call to complete (after having initiated the attempt to cancel it — see below);
- 8 waiting for the termination of dependent tasks;
- 9 the execution of an Initialize procedure as the last step of the default initialization of a controlled object;
- 10 the execution of a Finalize procedure as part of the finalization of a controlled object;
- 11 an assignment operation to an object with a controlled part.

12 [The last three of these are discussed further in 7.6.]

12.a **Reason:** Deferring abort during Initialize and finalization allows, for example, the result of an allocator performed in an Initialize operation to be assigned into an access object without being interrupted in the middle, which would cause storage leaks. For an object with several controlled parts, each individual Initialize is abort-deferred. Note that there is generally no semantic difference between making each Finalize abort-deferred, versus making a group of them abort-deferred, because if the task gets aborted, the first thing it will do is complete any remaining finalizations. Individual objects are finalized prior to an assignment operation (if nonlimited controlled) and as part of Unchecked_Deallocation.

12.b **Ramification:** Abort is deferred during the entire assignment operation to an object with a controlled part, even if only some subcomponents are controlled. Note that this says "assignment operation," not "assignment_statement." Explicit calls to Initialize, Finalize, or Adjust are not abort-deferred.

13 When a master is aborted, all tasks that depend on that master are aborted.

14 {*unspecified* [partial]} The order in which tasks become abnormal as the result of an abort_statement or the abort of a sequence_of_statements is not specified by the language.

15 If the execution of an entry call is aborted, an immediate attempt is made to cancel the entry call (see 9.5.3). If the execution of a construct is aborted at a time when the execution is blocked, other than for an entry call, at a point that is outside the execution of an abort-deferred operation, then the execution of the construct completes immediately. For an abort due to an abort_statement, these immediate effects occur before the execution of the abort_statement completes. Other than for these immediate cases, the execution of a construct that is aborted does not necessarily complete before the abort_statement completes. However, the execution of the aborted construct completes no later than its next *abort completion point* (if any) that occurs outside of an abort-deferred operation; {*abort completion point*} the following are abort completion points for an execution:

- 16 the point where the execution initiates the activation of another task;
- 17 the end of the activation of a task;
- 18 the start or end of the execution of an entry call, accept_statement, delay_statement, or abort_statement;

18.a **Ramification:** Although the abort completion point doesn't occur until the end of the entry call or delay_statement, these operations might be cut short because an abort attempts to cancel them.

- 19 the start of the execution of a select_statement, or of the sequence_of_statements of an exception_handler.

19.a **Reason:** The start of an exception_handler is considered an abort completion point simply because it is easy for an implementation to check at such points.

19.b **Implementation Note:** Implementations may of course check for abort more often than at each abort completion point; ideally, a fully preemptive implementation of abort will be provided. If preemptive abort is not supported in a given environment, then supporting the checking for abort as part of subprogram calls and loop iterations might be a useful option.

Bounded (Run-Time) Errors

20 {*bounded error (cause)* [partial]} An attempt to execute an asynchronous_select as part of the execution of an abort-deferred operation is a bounded error. Similarly, an attempt to create a task that depends on a master that is included entirely within the execution of an abort-deferred operation is a bounded error. {*Program_Error (raised by failure of run-time check)*} In both cases, Program_Error is raised if the error is detected by the implementation; otherwise the operations proceed as they would outside an abort-deferred operation, except that an abort of the abortable_part or the created task might or might not have an effect.

20.a **Reason:** An asynchronous_select relies on an abort of the abortable_part to effect the asynchronous transfer of control. For an asynchronous_select within an abort-deferred operation, the abort might have no effect.

20.b Creating a task dependent on a master included within an abort-deferred operation is considered an error, because such tasks could be aborted while the abort-deferred operation was still progressing, undermining the purpose of abort-deferral. Alternatively, we could say that such tasks are abort-deferred for their entire execution, but that seems too easy to abuse. Note that task creation is already a bounded error in protected actions, so this additional rule only applies to local task creation as part of Initialize, Finalize, or Adjust.

Erroneous Execution

21 {*normal state of an object* [partial]} {*abnormal state of an object* [partial]} {*disruption of an assignment*} {*erroneous execution (cause)* [partial]} If an assignment operation completes prematurely due to an abort, the assignment is said to be *disrupted*; the target of the assignment or its parts can become abnormal, and certain subsequent uses of the object can be erroneous, as explained in 13.9.1.

NOTES

22 41 An abort_statement should be used only in situations requiring unconditional termination.

23 42 A task is allowed to abort any task it can name, including itself.

24 43 Additional requirements associated with abort are given in D.6, “Preemptive Abort”.

Wording Changes from Ada 83

24.a This clause has been rewritten to accommodate the concept of aborting the execution of a construct, rather than just of a task.

9.9 Task and Entry Attributes

Dynamic Semantics

1 For a prefix T that is of a task type [(after any implicit dereference)], the following attributes are defined:

2 T'Callable Yields the value True when the task denoted by T is *callable*, and False otherwise; {*task state (callable)* [partial]} {*callable*} a task is callable unless it is completed or abnormal. The value of this attribute is of the predefined type Boolean.

3 T'Terminated
Yields the value True if the task denoted by T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean.

4 For a prefix E that denotes an entry of a task or protected unit, the following attribute is defined. This attribute is only allowed within the body of the task or protected unit, but excluding, in the case of an entry of a task unit, within any program unit that is, itself, inner to the body of the task unit.

E'Count Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type *universal_integer*. 5

NOTES

44 For the Count attribute, the entry can be either a single entry or an entry of a family. The name of the entry or entry family can be either a direct_name or an expanded name. 6

45 Within task units, algorithms interrogating the attribute E'Count should take precautions to allow for the increase of the value of this attribute for incoming entry calls, and its decrease, for example with timed_entry_calls. Also, a conditional_entry_call may briefly increase this value, even if the conditional call is not accepted. 7

46 Within protected units, algorithms interrogating the attribute E'Count in the entry_barrier for the entry E should take precautions to allow for the evaluation of the condition of the barrier both before and after queuing a given caller. 8

9.10 Shared Variables

Static Semantics

1 {*shared variable (protection of)*} {*independently addressable*} If two different objects, including nonoverlapping parts of the same object, are *independently addressable*, they can be manipulated concurrently by two different tasks without synchronization. Normally, any two nonoverlapping objects are independently addressable. However, if packing, record layout, or Component_Size is specified for a given composite object, then it is implementation defined whether or not two nonoverlapping parts of that composite object are independently addressable.

1.a **Implementation defined:** Whether or not two nonoverlapping parts of a composite object are independently addressable, in the case where packing, record layout, or Component_Size is specified for the object.

1.b **Implementation Note:** Independent addressability is the only high level semantic effect of a pragma Pack. If two objects are independently addressable, the implementation should allocate them in such a way that each can be written by the hardware without writing the other. For example, unless the user asks for it, it is generally not feasible to choose a bit-packed representation on a machine without an atomic bit field insertion instruction, because there might be tasks that update neighboring subcomponents concurrently, and locking operations on all subcomponents is generally not a good idea.

1.c Even if packing or one of the other above-mentioned aspects is specified, subcomponents should still be updated independently if the hardware efficiently supports it.

Dynamic Semantics

2 [Separate tasks normally proceed independently and concurrently with one another. However, task interactions can be used to synchronize the actions of two or more tasks to allow, for example, meaningful communication by the direct updating and reading of variables shared between the tasks.] The actions of two different tasks are synchronized in this sense when an action of one task *signals* an action of the other task; {*signal (as defined between actions)*} an action A1 is defined to signal an action A2 under the following circumstances:

- 3 If A1 and A2 are part of the execution of the same task, and the language rules require A1 to be performed before A2;
- 4 If A1 is the action of an activator that initiates the activation of a task, and A2 is part of the execution of the task that is activated;
- 5 If A1 is part of the activation of a task, and A2 is the action of waiting for completion of the activation;
- 6 If A1 is part of the execution of a task, and A2 is the action of waiting for the termination of the task;
- 6.1/1 {*8652/0031*} {*AI95-00118-01*} If A1 is the termination of a task T, and A2 is either the evaluation of the expression T'Terminated or a call to Ada.Task_Identification.Is_Terminated with an actual parameter that identifies T (see C.7.1);

7 • If A1 is the action of issuing an entry call, and A2 is part of the corresponding execution of the appropriate entry_body or accept_statement.

7.a **Ramification:** Evaluating the entry_index of an accept_statement is not synchronized with a corresponding entry call, nor is evaluating the entry barrier of an entry_body.

8 • If A1 is part of the execution of an accept_statement or entry_body, and A2 is the action of returning from the corresponding entry call;

9 • If A1 is part of the execution of a protected procedure body or entry_body for a given protected object, and A2 is part of a later execution of an entry_body for the same protected object;

9.a **Reason:** The underlying principle here is that for one action to "signal" a second, the second action has to follow a potentially blocking operation, whose blocking is dependent on the first action in some way. Protected procedures are not potentially blocking, so they can only be "signalers," they cannot be signaled.

9.b **Ramification:** Protected subprogram calls are not defined to signal one another, which means that such calls alone cannot be used to synchronize access to shared data outside of a protected object.

9.c **Reason:** The point of this distinction is so that on multiprocessors with inconsistent caches, the caches only need to be refreshed at the beginning of an entry body, and forced out at the end of an entry body or protected procedure that leaves an entry open. Protected function calls, and protected subprogram calls for entryless protected objects do not require full cache consistency. Entryless protected objects are intended to be treated roughly like atomic objects — each operation is indivisible with respect to other operations (unless both are reads), but such operations cannot be used to synchronize access to other nonvolatile shared variables.

10 • If A1 signals some action that in turn signals A2.

Erroneous Execution

11 {*erroneous execution (cause)* [partial]} Given an action of assigning to an object, and an action of reading or updating a part of the same object (or of a neighboring object if the two are not independently addressable), then the execution of the actions is erroneous unless the actions are *sequential*. {*sequential (actions)*} Two actions are sequential if one of the following is true:

12 • One action signals the other;

13 • Both actions occur as part of the execution of the same task;

13.a **Reason:** Any two actions of the same task are sequential, even if one does not signal the other because they can be executed in an "arbitrary" (but necessarily equivalent to some "sequential") order.

14 • Both actions occur as part of protected actions on the same protected object, and at most one of the actions is part of a call on a protected function of the protected object.

14.a **Reason:** Because actions within protected actions do not always imply signaling, we have to mention them here explicitly to make sure that actions occurring within different protected actions of the same protected object are sequential with respect to one another (unless both are part of calls on protected functions).

14.b **Ramification:** It doesn't matter whether or not the variable being assigned is actually a subcomponent of the protected object; globals can be safely updated from within the bodies of protected procedures or entries.

15 A pragma Atomic or Atomic_Components may also be used to ensure that certain reads and updates are sequential — see C.6.

15.a **Ramification:** If two actions are "sequential" it is known that their executions don't overlap in time, but it is not necessarily specified which occurs first. For example, all actions of a single task are sequential, even though the exact order of execution is not fully specified for all constructs.

15.b **Discussion:** Note that if two assignments to the same variable are sequential, but neither signals the other, then the program is not erroneous, but it is not specified which assignment ultimately prevails. Such a situation usually corresponds to a programming mistake, but in some (rare) cases, the order makes no difference, and for this reason this situation is not considered erroneous nor even a bounded error. In Ada 83, this was considered an "incorrect order dependence" if the "effect" of the program was affected, but "effect" was never fully defined. In Ada 95, this situation represents a potential nonportability, and a friendly compiler might want to warn the programmer about the situation, but it is not considered an error. An example where this would come up would be in gathering statistics as part of referencing some information, where the assignments associated with statistics gathering don't need to be ordered since they are just accumulating aggregate counts, sums, products, etc.

Wording Changes from Ada 95

{*8652/0031*} {*AI95-00118-01*} **Corrigendum:** Clarified that a task T2 can rely on values of variables that are updated by another task T1, if task T2 first verifies that T1'Terminated is True. 15.c/2

9.11 Example of Tasking and Synchronization

Examples

The following example defines a buffer protected object to smooth variations between the speed of output of a producing task and the speed of input of some consuming task. For instance, the producing task might have the following structure: 1

```
task Producer;                                                    2
```

```
{AI95-00433-01} task body Producer is                             3/2
   Person : Person_Name; -- see 3.10.1
begin
   loop
      ... -- simulate arrival of the next customer
      Buffer.Append_Wait(Person);
      exit when Person = null;
   end loop;
end Producer;
```

and the consuming task might have the following structure: 4

```
task Consumer;                                                    5
```

```
{AI95-00433-01} task body Consumer is                             6/2
   Person : Person_Name;
begin
   loop
      Buffer.Remove_First_Wait(Person);
      exit when Person = null;
      ... -- simulate serving a customer
   end loop;
end Consumer;
```

{*AI95-00433-01*} The buffer object contains an internal array of person names managed in a round-robin fashion. The array has two indices, an In_Index denoting the index for the next input person name and an Out_Index denoting the index for the next output person name. 7/2

{*AI95-00433-01*} The Buffer is defined as an extension of the Synchronized_Queue interface (see 3.9.4), and as such promises to implement the abstraction defined by that interface. By doing so, the Buffer can be passed to the Transfer class-wide operation defined for objects of a type covered by Queue'Class. 7.1/2

```
{AI95-00433-01} protected Buffer is new Synchronized_Queue with  -- see 3.9.4   8/2
   entry Append_Wait(Person : in Person_Name);
   entry Remove_First_Wait(Person : out Person_Name);
   function Cur_Count return Natural;
   function Max_Count return Natural;
   procedure Append(Person : in Person_Name);
   procedure Remove_First(Person : out Person_Name);
private
   Pool      : Person_Name_Array(1 .. 100);
   Count     : Natural := 0;
   In_Index, Out_Index : Positive := 1;
end Buffer;
```

```
{AI95-00433-01} protected body Buffer is                          9/2
   entry Append_Wait(Person : in Person_Name)
      when Count < Pool'Length is
   begin
      Append(Person);
   end Append_Wait;
```

9.1/2
```
{AI95-00433-01}    procedure Append(Person : in Person_Name) is
   begin
      if Count = Pool'Length then
         raise Queue_Error with "Buffer Full";  -- see 11.3
      end if;
      Pool(In_Index) := Person;
      In_Index       := (In_Index mod Pool'Length) + 1;
      Count          := Count + 1;
   end Append;
```

10/2
```
{AI95-00433-01}    entry Remove_First_Wait(Person : out Person_Name)
      when Count > 0 is
   begin
      Remove_First(Person);
   end Remove_First_Wait;
```

11/2
```
{AI95-00433-01}    procedure Remove_First(Person : out Person_Name) is
   begin
      if Count = 0 then
         raise Queue_Error with "Buffer Empty"; -- see 11.3
      end if;
      Person    := Pool(Out_Index);
      Out_Index := (Out_Index mod Pool'Length) + 1;
      Count     := Count - 1;
   end Remove_First;
```

12/2
```
{AI95-00433-01}    function Cur_Count return Natural is
   begin
       return Buffer.Count;
   end Cur_Count;
```

13/2
```
{AI95-00433-01}    function Max_Count return Natural is
   begin
       return Pool'Length;
   end Max_Count;
end Buffer;
```

Section 10: Program Structure and Compilation Issues

[The overall structure of programs and the facilities for separate compilation are described in this section. A *program* is a set of *partitions*, each of which may execute in a separate address space, possibly on a separate computer. 1

Glossary entry: {*Program*} A *program* is a set of *partitions*, each of which may execute in a separate address space, possibly on a separate computer. A partition consists of a set of library units. 1.a

Glossary entry: {*Partition*} A *partition* is a part of a program. Each partition consists of a set of library units. Each partition may run in a separate address space, possibly on a separate computer. A program may contain just one partition. A distributed program typically contains multiple partitions, which can execute concurrently. 1.b

{*library unit (informal introduction)*} {*library_item (informal introduction)*} {*library (informal introduction)*} As explained below, a partition is constructed from *library units*. Syntactically, the declaration of a library unit is a library_item, as is the body of a library unit. An implementation may support a concept of a *program library* (or simply, a "library"), which contains library_items and their subunits. {*program library: See library*} Library units may be organized into a hierarchy of children, grandchildren, and so on.] 2

This section has two clauses: 10.1, "Separate Compilation" discusses compile-time issues related to separate compilation. 10.2, "Program Execution" discusses issues related to what is traditionally known as "link time" and "run time" — building and executing partitions. 3

Language Design Principles

{*avoid overspecifying environmental issues*} We should avoid specifying details that are outside the domain of the language itself. The standard is intended (at least in part) to promote portability of Ada programs at the source level. It is not intended to standardize extra-language issues such as how one invokes the compiler (or other tools), how one's source is represented and organized, version management, the format of error messages, etc. 3.a

{*safe separate compilation*} {*separate compilation (safe)*} The rules of the language should be enforced even in the presence of separate compilation. Using separate compilation should not make a program less safe. 3.b

{*legality determinable via semantic dependences*} It should be possible to determine the legality of a compilation unit by looking only at the compilation unit itself and the compilation units upon which it depends semantically. As an example, it should be possible to analyze the legality of two compilation units in parallel if they do not depend semantically upon each other. 3.c

On the other hand, it may be necessary to look outside that set in order to generate code — this is generally true for generic instantiation and inlining, for example. Also on the other hand, it is generally necessary to look outside that set in order to check Post-Compilation Rules. 3.d

See also the "generic contract model" Language Design Principle of 12.3, "Generic Instantiation". 3.e

Wording Changes from Ada 83

The section organization mentioned above is different from that of RM83. 3.f

10.1 Separate Compilation

[{*separate compilation*} {*compilation (separate)*} {*Program unit*} [Glossary Entry]A *program unit* is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units. 1

{*Compilation unit*} [Glossary Entry]The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation_units. A compilation_unit contains either the declaration, the body, or a renaming of a program unit.] The representation for a compilation is implementation-defined. 2

Implementation defined: The representation for a compilation. 2.a

2.b **Ramification:** Some implementations might choose to make a compilation be a source (text) file. Others might allow multiple source files to be automatically concatenated to form a single compilation. Others still may represent the source in a nontextual form such as a parse tree. Note that the RM95 does not even define the concept of a source file.

2.c Note that a protected subprogram is a subprogram, and therefore a program unit. An instance of a generic unit is a program unit.

2.d A protected entry is a program unit, but protected entries cannot be separately compiled.

3 {*Library unit*} [Glossary Entry]A library unit is a separately compiled program unit, and is always a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. {*subsystem*} A root library unit, together with its children and grandchildren and so on, form a *subsystem*.

Implementation Permissions

4 An implementation may impose implementation-defined restrictions on compilations that contain multiple compilation_units.

4.a **Implementation defined:** Any restrictions on compilations that contain multiple compilation_units.

4.b **Discussion:** For example, an implementation might disallow a compilation that contains two versions of the same compilation unit, or that contains the declarations for library packages P1 and P2, where P1 precedes P2 in the compilation but P1 has a with_clause that mentions P2.

Wording Changes from Ada 83

4.c The interactions between language issues and environmental issues are left open in Ada 95. The environment concept is new. In Ada 83, the concept of the program library, for example, appeared to be quite concrete, although the rules had no force, since implementations could get around them simply by defining various mappings from the concept of an Ada program library to whatever data structures were actually stored in support of separate compilation. Indeed, implementations were encouraged to do so.

4.d In RM83, it was unclear which was the official definition of “program unit.” Definitions appeared in RM83-5, 6, 7, and 9, but not 12. Placing it here seems logical, since a program unit is sort of a potential compilation unit.

10.1.1 Compilation Units - Library Units

1 [A library_item is a compilation unit that is the declaration, body, or renaming of a library unit. Each library unit (except Standard) has a *parent unit*, which is a library package or generic library package.] {*child (of a library unit)*} A library unit is a *child* of its parent unit. The *root* library units are the children of the predefined library package Standard.

1.a **Ramification:** Standard is a library unit.

Syntax

2 compilation ::= {compilation_unit}

3
```
compilation_unit ::=
    context_clause library_item
  | context_clause subunit
```

4
```
library_item ::= [private] library_unit_declaration
  | library_unit_body
  | [private] library_unit_renaming_declaration
```

5
```
library_unit_declaration ::=
     subprogram_declaration     | package_declaration
   | generic_declaration        | generic_instantiation
```

library_unit_renaming_declaration ::=
  package_renaming_declaration
 | generic_renaming_declaration
 | subprogram_renaming_declaration 6

library_unit_body ::= subprogram_body | package_body 7

parent_unit_name ::= name 8

{*AI95-00397-01*} An overriding_indicator is not allowed in a subprogram_declaration, generic_instantiation, or subprogram_renaming_declaration that declares a library unit. 8.1/2

Reason: All of the listed items syntactically include overriding_indicator, but a library unit can never override anything. A majority of the ARG thought that allowing **not overriding** in that case would be confusing instead of helpful. 8.a.1/2

{*library unit*} {*library* [partial]} A *library unit* is a program unit that is declared by a library_item. When a program unit is a library unit, the prefix "library" is used to refer to it (or "generic library" if generic), as well as to its declaration and body, as in "library procedure", "library package_body", or "generic library package". {*compilation unit*} The term *compilation unit* is used to refer to a compilation_unit. When the meaning is clear from context, the term is also used to refer to the library_item of a compilation_unit or to the proper_body of a subunit [(that is, the compilation_unit without the context_clause and the **separate** (parent_unit_name))]. 9

Discussion: In this example: 9.a

9.b
```
with Ada.Text_IO;
package P is
    ...
end P;
```

the term "compilation unit" can refer to this text: "**with** Ada.Text_IO; **package** P **is** ... **end** P;" or to this text: "**package** P **is** ... **end** P;". We use this shorthand because it corresponds to common usage. 9.c

We like to use the word "unit" for declaration-plus-body things, and "item" for declaration or body separately (as in declarative_item). The terms "compilation_unit," "compilation unit," and "subunit" are exceptions to this rule. We considered changing "compilation_unit," "compilation unit" to "compilation_item," "compilation item," respectively, but we decided not to. 9.d

{*parent declaration (of a library_item)*} {*parent declaration (of a library unit)*} The *parent declaration* of a library_item (and of the library unit) is the declaration denoted by the parent_unit_name, if any, of the defining_program_unit_name of the library_item. {*root library unit*} If there is no parent_unit_name, the parent declaration is the declaration of Standard, the library_item is a *root* library_item, and the library unit (renaming) is a *root* library unit (renaming). The declaration and body of Standard itself have no parent declaration. {*parent unit (of a library unit)*} The *parent unit* of a library_item or library unit is the library unit declared by its parent declaration. 10

Discussion: The declaration and body of Standard are presumed to exist from the beginning of time, as it were. There is no way to actually write them, since there is no syntactic way to indicate lack of a parent. An attempt to compile a package Standard would result in Standard.Standard. 10.a

Reason: Library units (other than Standard) have "parent declarations" and "parent units". Subunits have "parent bodies". We didn't bother to define the other possibilities: parent body of a library unit, parent declaration of a subunit, parent unit of a subunit. These are not needed, and might get in the way of a correct definition of "child." 10.b

[The children of a library unit occur immediately within the declarative region of the declaration of the library unit.] {*ancestor (of a library unit)*} The *ancestors* of a library unit are itself, its parent, its parent's parent, and so on. [(Standard is an ancestor of every library unit.)] {*descendant*} The *descendant* relation is the inverse of the ancestor relation. 11

Reason: These definitions are worded carefully to avoid defining subunits as children. Only library units can be children. 11.a

We use the unadorned term "ancestors" here to concisely define both "ancestor unit" and "ancestor declaration." 11.b

12 {*public library unit*} {*public declaration of a library unit*} {*private library unit*} {*private declaration of a library unit*} A library_unit_declaration or a library_unit_renaming_declaration is *private* if the declaration is immediately preceded by the reserved word **private**; it is otherwise *public*. A library unit is private or public according to its declaration. {*public descendant (of a library unit)*} The *public descendants* of a library unit are the library unit itself, and the public descendants of its public children. {*private descendant (of a library unit)*} Its other descendants are *private descendants*.

12.a **Discussion:** The first concept defined here is that a library_item is either public or private (not in relation to anything else — it's just a property of the library unit). The second concept is that a library_item is a public descendant or private descendant *of a given ancestor*. A given library_item can be a public descendant of one of its ancestors, but a private descendant of some other ancestor.

12.b A subprogram declared by a subprogram_body (as opposed to a subprogram_declaration) is always public, since the syntax rules disallow the reserved word **private** on a body.

12.c Note that a private library unit is a *public* descendant of itself, but a *private* descendant of its parent. This is because it is visible outside itself — its privateness means that it is not visible outside its parent.

12.d Private children of Standard are legal, and follow the normal rules. It is intended that implementations might have some method for taking an existing environment, and treating it as a package to be "imported" into another environment, treating children of Standard in the imported environment as children of the imported package.

12.e **Ramification:** Suppose we have a public library unit A, a private library unit A.B, and a public library unit A.B.C. A.B.C is a public descendant of itself and of A.B, but a private descendant of A; since A.B is private to A, we don't allow A.B.C to escape outside A either. This is similar to the situation that would occur with physical nesting, like this:

12.f
```
package A is
private
    package B is
        package C is
        end C;
    private
    end B;
end A;
```

12.g Here, A.B.C is visible outside itself and outside A.B, but not outside A. (Note that this example is intended to illustrate the visibility of program units from the outside; the visibility within child units is not quite identical to that of physically nested units, since child units are nested after their parent's declaration.)

12.1/2 {*AI95-00217-06*} For each library package_declaration in the environment, there is an implicit declaration of a *limited view* of that library package.{*limited view*} The limited view of a package contains:

12.2/2
- {*AI95-00217-06*} For each nested package_declaration, a declaration of the limited view of that package, with the same defining_program_unit_name.

12.3/2
- {*AI95-00217-06*} {*AI95-00326-01*} For each type_declaration in the visible part, an incomplete view of the type; if the type_declaration is tagged, then the view is a tagged incomplete view.

12.g.1/2 **Discussion:** {*AI95-00217-06*} The implementation model of a limited view is that it can be determined solely from the syntax of the source of the unit, without any semantic analysis. That allows it to be created without the semantic dependences of a full unit, which is necessary for it to break mutual dependences of units.

12.g.2/2 **Ramification:** The limited view does not include package instances and their contents. Semantic analysis of a unit (and dependence on its with_clauses) would be needed to determine the contents of an instance.

12.4/2 The limited view of a library package_declaration is private if that library package_declaration is immediately preceded by the reserved word **private**.

12.5/2 [There is no syntax for declaring limited views of packages, because they are always implicit.] The implicit declaration of a limited view of a library package [is not the declaration of a library unit (the library package_declaration is); nonetheless, it] is a library_item. The implicit declaration of the limited view of a library package forms an (implicit) compilation unit whose context_clause is empty.

12.6/2 A library package_declaration is the completion of the declaration of its limited view.

12.h/2 **To be honest:** This is notwithstanding the rule in 3.11.1 that says that implicit declarations don't have completions.

Reason: This rule explains where to find the completions of the incomplete views defined by the limited view. 12.i/2

Legality Rules

The parent unit of a library_item shall be a [library] package or generic [library] package. 13

If a defining_program_unit_name of a given declaration or body has a parent_unit_name, then the given declaration or body shall be a library_item. The body of a program unit shall be a library_item if and only if the declaration of the program unit is a library_item. In a library_unit_renaming_declaration, the [(old)] name shall denote a library_item. 14

Discussion: We could have allowed nested program units to be children of other program units; their semantics would make sense. We disallow them to keep things simpler and because they wouldn't be particularly useful. 14.a

{*AI95-00217-06*} A parent_unit_name [(which can be used within a defining_program_unit_name of a library_item and in the **separate** clause of a subunit)], and each of its prefixes, shall not denote a renaming_declaration. [On the other hand, a name that denotes a library_unit_renaming_declaration is allowed in a nonlimited_with_clause and other places where the name of a library unit is allowed.] 15/2

If a library package is an instance of a generic package, then every child of the library package shall either be itself an instance or be a renaming of a library unit. 16

Discussion: A child of an instance of a given generic unit will often be an instance of a (generic) child of the given generic unit. This is not required, however. 16.a

Reason: Instances are forbidden from having noninstance children for two reasons: 16.b

1. We want all source code that can depend on information from the private part of a library unit to be inside the "subsystem" rooted at the library unit. If an instance of a generic unit were allowed to have a noninstance as a child, the source code of that child might depend on information from the private part of the generic unit, even though it is outside the subsystem rooted at the generic unit. 16.c
2. Disallowing noninstance children simplifies the description of the semantics of children of generic packages. 16.d

A child of a generic library package shall either be itself a generic unit or be a renaming of some other child of the same generic unit. The renaming of a child of a generic package shall occur only within the declarative region of the generic package. 17

A child of a parent generic package shall be instantiated or renamed only within the declarative region of the parent generic. 18

{*AI95-00331-01*} For each child *C* of some parent generic package *P*, there is a corresponding declaration *C* nested immediately within each instance of *P*. For the purposes of this rule, if a child *C* itself has a child *D*, each corresponding declaration for *C* has a corresponding child *D*. [The corresponding declaration for a child within an instance is visible only within the scope of a with_clause that mentions the (original) child generic unit.] 19/2

Implementation Note: Within the child, like anything nested in a generic unit, one can make up-level references to the current instance of its parent, and thereby gain access to the formal parameters of the parent, to the types declared in the parent, etc. This “nesting” model applies even within the generic_formal_part of the child, as it does for a generic child of a nongeneric unit. 19.a

Ramification: Suppose P is a generic library package, and P.C is a generic child of P. P.C can be instantiated inside the declarative region of P. Outside P, P.C can be mentioned only in a with_clause. Conceptually, an instance I of P is a package that has a nested generic unit called I.C. Mentioning P.C in a with_clause allows I.C to be instantiated. I need not be a library unit, and the instantiation of I.C need not be a library unit. If I is a library unit, and an instance of I.C is a child of I, then this instance has to be called something other than C. 19.b

A library subprogram shall not override a primitive subprogram. 20

Reason: This prevents certain obscure anomalies. For example, if a library subprogram were to override a subprogram declared in its parent package, then in a compilation unit that depends *in*directly on the library subprogram, the library subprogram could hide the overridden operation from all visibility, but the library subprogram itself would not be visible. 20.a

Note that even without this rule, such subprograms would be illegal for tagged types, because of the freezing rules. 20.b

21 The defining name of a function that is a compilation unit shall not be an operator_symbol.

21.a **Reason:** Since overloading is not permitted among compilation units, it seems unlikely that it would be useful to define one as an operator. Note that a subunit could be renamed within its parent to be an operator.

Static Semantics

22 A subprogram_renaming_declaration that is a library_unit_renaming_declaration is a renaming-as-declaration, not a renaming-as-body.

23 [There are two kinds of dependences among compilation units:]

24 • [The *semantic dependences* (see below) are the ones needed to check the compile-time rules across compilation unit boundaries; a compilation unit depends semantically on the other compilation units needed to determine its legality. The visibility rules are based on the semantic dependences.

25 • The *elaboration dependences* (see 10.2) determine the order of elaboration of library_items.]

25.a **Discussion:** Don't confuse these kinds of dependences with the run-time dependences among tasks and masters defined in 9.3, “Task Dependence - Termination of Tasks”.

26/2 {*AI95-00217-06*} {*semantic dependence (of one compilation unit upon another)*} {*dependence (semantic)*} A library_item depends semantically upon its parent declaration. A subunit depends semantically upon its parent body. A library_unit_body depends semantically upon the corresponding library_unit_declaration, if any. The declaration of the limited view of a library package depends semantically upon the declaration of the limited view of its parent. The declaration of a library package depends semantically upon the declaration of its limited view. A compilation unit depends semantically upon each library_item mentioned in a with_clause of the compilation unit. In addition, if a given compilation unit contains an attribute_reference of a type defined in another compilation unit, then the given compilation unit depends semantically upon the other compilation unit. The semantic dependence relationship is transitive.

26.a **Discussion:** The “if any” in the third sentence is necessary because library subprograms are not required to have a subprogram_declaration.

26.b **To be honest:** If a given compilation unit contains a choice_parameter_specification, then the given compilation unit depends semantically upon the declaration of Ada.Exceptions.

26.c If a given compilation unit contains a pragma with an argument of a type defined in another compilation unit, then the given compilation unit depends semantically upon the other compilation unit.

26.d **Discussion:** For example, a compilation unit containing X'Address depends semantically upon the declaration of package System.

26.e For the Address attribute, this fixes a hole in Ada 83. Note that in almost all cases, the dependence will need to exist due to with_clauses, even without this rule. Hence, the rule has very little effect on programmers.

26.f Note that the semantic dependence does not have the same effect as a with_clause; in order to denote a declaration in one of those packages, a with_clause will generally be needed.

26.g Note that no special rule is needed for an attribute_definition_clause, since an expression after **use** will require semantic dependence upon the compilation unit containing the type_declaration of interest.

26.h/2 {*AI95-00217-06*} Unlike a full view of a package, a limited view does not depend semantically on units mentioned in with_clauses of the compilation_unit that defines the package. Formally, this is achieved by saying that the limited view has an empty context_clause. This is necessary so that they can be useful for their intended purpose: allowing mutual dependences between packages. The lack of semantic dependence limits the contents of a limited view to the items that can be determined solely from the syntax of the source of the package, without any semantic analysis. That allows it to be created without the semantic dependences of a full package.

Dynamic Semantics

26.1/2 {*AI95-00217-06*} The elaboration of the declaration of the limited view of a package has no effect.

NOTES

27 1 A simple program may consist of a single compilation unit. A compilation need not have any compilation units; for example, its text can consist of pragmas.

Ramification: Such pragmas cannot have any arguments that are names, by a previous rule of this subclause. A compilation can even be entirely empty, which is probably not useful. 27.a

Some interesting properties of the three kinds of dependence: The elaboration dependences also include the semantic dependences, except that subunits are taken together with their parents. The semantic dependences partly determine the order in which the compilation units appear in the environment at compile time. At run time, the order is partly determined by the elaboration dependences. 27.b

The model whereby a child is inside its parent's declarative region, after the parent's declaration, as explained in 8.1, has the following ramifications: 27.c

- The restrictions on “early” use of a private type (RM83-7.4.1(4)) or a deferred constant (RM83-7.4.3(2)) do not apply to uses in child units, because they follow the full declaration. 27.d
- A library subprogram is never primitive, even if its profile includes a type declared immediately within the parent's package_specification, because the child is not declared immediately within the same package_specification as the type (so it doesn't declare a new primitive subprogram), and because the child is forbidden from overriding an old primitive subprogram. It is immediately within the same declarative region, but not the same package_specification. Thus, for a tagged type, it is not possible to call a child subprogram in a dispatching manner. (This is also forbidden by the freezing rules.) Similarly, it is not possible for the user to declare primitive subprograms of the types declared in the declaration of Standard, such as Integer (even if the rules were changed to allow a library unit whose name is an operator symbol). 27.e
- When the parent unit is “used” the simple names of the with'd child units are directly visible (see 8.4, “Use Clauses”). 27.f
- When a parent body with's its own child, the defining name of the child is directly visible, and the parent body is not allowed to include a declaration of a homograph of the child unit immediately within the declarative_part of the body (RM83-8.3(17)). 27.g

Note that “declaration of a library unit” is different from “library_unit_declaration” — the former includes subprogram_body. Also, we sometimes really mean “declaration of a view of a library unit”, which includes library_-unit_renaming_declarations. 27.h

The visibility rules generally imply that the renamed view of a library_unit_renaming_declaration has to be mentioned in a with_clause of the library_unit_renaming_declaration. 27.i

To be honest: The real rule is that the renamed library unit has to be visible in the library_unit_renaming_declaration. 27.j

Reason: In most cases, “has to be visible” means there has to be a with_clause. However, it is possible in obscure cases to avoid the need for a with_clause; in particular, a compilation unit such as “**package** P.Q **renames** P;” is legal with no with_clauses (though not particularly interesting). ASCII is physically nested in Standard, and so is not a library unit, and cannot be renamed as a library unit. 27.k

2 The designator of a library function cannot be an operator_symbol, but a nonlibrary renaming_declaration is allowed to rename a library function as an operator. Within a partition, two library subprograms are required to have distinct names and hence cannot overload each other. However, renaming_declarations are allowed to define overloaded names for such subprograms, and a locally declared subprogram is allowed to overload a library subprogram. The expanded name Standard.L can be used to denote a root library unit L (unless the declaration of Standard is hidden) since root library unit declarations occur immediately within the declarative region of package Standard. 28

Examples

Examples of library units: 29

30
```
package Rational_Numbers.IO is  -- public child of Rational_Numbers, see 7.1
   procedure Put(R : in  Rational);
   procedure Get(R : out Rational);
end Rational_Numbers.IO;
```

31
```
private procedure Rational_Numbers.Reduce(R : in out Rational);
                                    -- private child of Rational_Numbers
```

32
```
with Rational_Numbers.Reduce;   -- refer to a private child
package body Rational_Numbers is
   ...
end Rational_Numbers;
```

33
```
with Rational_Numbers.IO; use Rational_Numbers;
with Ada.Text_io;                  -- see A.10
procedure Main is                  -- a root library procedure
   R : Rational;
begin
   R := 5/3;                       -- construct a rational number, see 7.1
   Ada.Text_IO.Put("The answer is: ");
   IO.Put(R);
   Ada.Text_IO.New_Line;
end Main;
```

34
```
with Rational_Numbers.IO;
package Rational_IO renames Rational_Numbers.IO;
                                   -- a library unit renaming declaration
```

35 Each of the above library_items can be submitted to the compiler separately.

35.a **Discussion:** *Example of a generic package with children:*

35.b
```
generic
   type Element is private;
   with function Image(E : Element) return String;
package Generic_Bags is
   type Bag is limited private; -- A bag of Elements.
   procedure Add(B : in out Bag; E : Element);
   function Bag_Image(B : Bag) return String;
private
   type Bag is ...;
end Generic_Bags;
```

35.c
```
generic
package Generic_Bags.Generic_Iterators is
   ... -- various additional operations on Bags.
```

35.d
```
   generic
      with procedure Use_Element(E : in Element);
         -- Called once per bag element.
   procedure Iterate(B : in Bag);
end Generic_Bags.Generic_Iterators;
```

35.e A package that instantiates the above generic units:

35.f
```
with Generic_Bags;
with Generic_Bags.Generic_Iterators;
package My_Abstraction is
    type My_Type is ...;
    function Image(X : My_Type) return String;
    package Bags_Of_My_Type is new Generic_Bags(My_Type, Image);
    package Iterators_Of_Bags_Of_My_Type is new Bags_Of_My_Type.Generic_Iterators;
end My_Abstraction;
```

35.g In the above example, Bags_Of_My_Type has a nested generic unit called Generic_Iterators. The second with_clause makes that nested unit visible.

35.h Here we show how the generic body could depend on one of its own children:

35.i
```
with Generic_Bags.Generic_Iterators;
package body Generic_Bags is
   procedure Add(B : in out Bag; E : Element) is ... end Add;
```

35.j
```
   package Iters is new Generic_Iterators;
```

35.k
```
   function Bag_Image(B : Bag) return String is
      Buffer : String(1..10_000);
      Last : Integer := 0;
```

35.l
```
      procedure Append_Image(E : in Element) is
         Im : constant String := Image(E);
      begin
         if Last /= 0 then -- Insert a comma.
            Last := Last + 1;
            Buffer(Last) := ',';
         end if;
         Buffer(Last+1 .. Last+Im'Length) := Im;
         Last := Last + Im'Length;
      end Append_Image;
```

```
        procedure Append_All is new Iters.Iterate(Append_Image);
     begin
        Append_All(B);
        return Buffer(1..Last);
     end Bag_Image;
  end Generic_Bags;
```
35.m

Extensions to Ada 83

{*extensions to Ada 83*} The syntax rule for library_item is modified to allow the reserved word **private** before a library_unit_declaration. 35.n

Children (other than children of Standard) are new in Ada 95. 35.o

Library unit renaming is new in Ada 95. 35.p

Wording Changes from Ada 83

Standard is considered a library unit in Ada 95. This simplifies the descriptions, since it implies that the parent of each library unit is a library unit. (Standard itself has no parent, of course.) As in Ada 83, the language does not define any way to recompile Standard, since the name given in the declaration of a library unit is always interpreted in relation to Standard. That is, an attempt to compile a package Standard would result in Standard.Standard. 35.q

Extensions to Ada 95

{*AI95-00217-06*} {*extensions to Ada 95*} The concept of a limited view is new. Combined with limited_with_clauses (see 10.1.2), they facilitate construction of mutually recursive types in multiple packages. 35.r/2

Wording Changes from Ada 95

{*AI95-00331-01*} Clarified the wording so that a grandchild generic unit will work as expected. 35.s/2

10.1.2 Context Clauses - With Clauses

[A context_clause is used to specify the library_items whose names are needed within a compilation unit.] 1

Language Design Principles

{*one-pass context_clauses*} The reader should be able to understand a context_clause without looking ahead. Similarly, when compiling a context_clause, the compiler should not have to look ahead at subsequent context_items, nor at the compilation unit to which the context_clause is attached. (We have not completely achieved this.) 1.a

{*AI95-00217-06*} {*ripple effect*} A *ripple effect* occurs when the legality of a compilation unit could be affected by adding or removing an otherwise unneeded with_clause on some compilation unit on which the unit depends, directly or indirectly. We try to avoid ripple effects because they make understanding and maintenance more difficult. However, ripple effects can occur because of direct visibility (as in child units); this seems impossible to eliminate. The ripple effect for with_clauses is somewhat similar to the Beaujolais effect (see 8.4) for use_clauses, which we also try to avoid. 1.b/2

Syntax

context_clause ::= {context_item} 2

context_item ::= with_clause | use_clause 3

{*AI95-00217-06*} {*AI95-00262-01*} with_clause ::= limited_with_clause | nonlimited_with_clause 4/2

limited_with_clause ::= **limited** [**private**] **with** *library_unit_*name {, *library_unit_*name}; 4.1/2

nonlimited_with_clause ::= [**private**] **with** *library_unit_*name {, *library_unit_*name}; 4.2/2

Discussion: {*AI95-00217-06*} A limited_with_clause makes a limited view of a unit visible. 4.a/2

{*AI95-00262-01*} {*private with_clause*} A with_clause containing the reserved word **private** is called a *private with_clause*. It can be thought of as making items visible only in the private part, although it really makes items visible everywhere except the visible part. It can be used both for documentation purposes (to say that a unit is not used in the visible part), and to allow access to private units that otherwise would be prohibited. 4.b/2

Name Resolution Rules

5 {*scope (of a with_clause)*} The *scope* of a with_clause that appears on a library_unit_declaration or library_unit_renaming_declaration consists of the entire declarative region of the declaration[, which includes all children and subunits]. The scope of a with_clause that appears on a body consists of the body[, which includes all subunits].

5.a/2 **Discussion:** {*AI95-00262-01*} Suppose a non-private with_clause of a public library unit mentions one of its private siblings. (This is only allowed on the body of the public library unit.) We considered making the scope of that with_clause not include the visible part of the public library unit. (This would only matter for a subprogram_body, since those are the only kinds of body that have a visible part, and only if the subprogram_body completes a subprogram_declaration, since otherwise the with_clause would be illegal.) We did not put in such a rule for two reasons: (1) It would complicate the wording of the rules, because we would have to split each with_clause into pieces, in order to correctly handle "**with** P, Q;" where P is public and Q is private. (2) The conformance rules prevent any problems. It doesn't matter if a type name in the spec of the body denotes the completion of a private_type_declaration.

5.b A with_clause also affects visibility within subsequent use_clauses and pragmas of the same context_clause, even though those are not in the scope of the with_clause.

6/2 {*AI95-00217-06*} {*Term=[mentioned],Sec=[in a with_clause]*} {*with_clause (mentioned in)*} A library_item (and the corresponding library unit) is *named* {*named (in a with_clause)*} {*with_clause (named in)*} in a with_clause if it is denoted by a *library_unit_*name in the with_clause. A library_item (and the corresponding library unit) is *mentioned* in a with_clause if it is named in the with_clause or if it is denoted by a prefix in the with_clause.

6.a **Discussion:** With_clauses control the visibility of declarations or renamings of library units. Mentioning a root library unit in a with_clause makes its declaration directly visible. Mentioning a non-root library unit makes its declaration visible. See Section 8 for details.

6.b/2 {*AI95-00114-01*} Note that this rule implies that "**with** A.B.C;" is almost equivalent to "**with** A, A.B, A.B.C;". The reason for making a with_clause apply to all the ancestor units is to avoid "visibility holes" — situations in which an inner program unit is visible while an outer one is not. Visibility holes would cause semantic complexity and implementation difficulty. (This is not exactly equivalent because the latter with_clause names A and A.B, while the previous one does not. Whether a unit is "named" does not have any effect on visibility, however, so it is equivalent for visibility purposes.))

7 [Outside its own declarative region, the declaration or renaming of a library unit can be visible only within the scope of a with_clause that mentions it. The visibility of the declaration or renaming of a library unit otherwise follows from its placement in the environment.]

Legality Rules

8/2 {*AI95-00262-01*} If a with_clause of a given compilation_unit mentions a private child of some library unit, then the given compilation_unit shall be one of:

9/2 • {*AI95-00262-01*} the declaration, body, or subunit of a private descendant of that library unit;

10/2 • {*AI95-00220-01*} {*AI95-00262-01*} the body or subunit of a public descendant of that library unit, but not a subprogram body acting as a subprogram declaration (see 10.1.4); or

11/2 • {*AI95-00262-01*} the declaration of a public descendant of that library unit, in which case the with_clause shall include the reserved word **private**.

11.a/2 **Reason:** {*AI95-00262-01*} The purpose of this rule is to prevent a private child from being visible from outside the subsystem rooted at its parent. A private child can be semantically depended-on without violating this principle if it is used in a private with_clause.

11.b **Discussion:** This rule violates the one-pass context_clauses Language Design Principle. We rationalize this by saying that at least that Language Design Principle works for legal compilation units.

11.c Example:

11.d
```
package A is
end A;
```

11.e
```
package A.B is
end A.B;
```
11.f
```
private package A.B.C is
end A.B.C;
```
11.g
```
package A.B.C.D is
end A.B.C.D;
```
11.h
```
with A.B.C; -- (1)
private package A.B.X is
end A.B.X;
```
11.i
```
package A.B.Y is
end A.B.Y;
```
11.j
```
with A.B.C; -- (2)
package body A.B.Y is
end A.B.Y;
```
11.j.1/2
```
private with A.B.C; -- (3)
package A.B.Z is
end A.B.Z;
```

11.k/2 {*AI95-00262-01*} (1) is OK because it's a private child of A.B — it would be illegal if we made A.B.X a public child of A.B. (2) is OK because it's the body of a child of A.B. (3) is OK because it's a child of A.B, and it is a private with_clause. It would be illegal to say "**with** A.B.C;" on any library_item whose name does not start with "A.B". Note that mentioning A.B.C.D in a with_clause automatically mentions A.B.C as well, so "**with** A.B.C.D;" is illegal in the same places as "**with** A.B.C;".

11.l/2 **To be honest:** {*AI95-00262-01*} For the purposes of this rule, if a subprogram_body has no preceding subprogram_declaration, the subprogram_body should be considered a declaration and not a body. Thus, it is illegal for such a subprogram_body to mention one of its siblings in a non-private with_clause if the sibling is a private library unit.

12/2 {*AI95-00262-01*} A name denoting a library item that is visible only due to being mentioned in one or more with_clauses that include the reserved word **private** shall appear only within:

- 13/2 a private part;
- 14/2 a body, but not within the subprogram_specification of a library subprogram body;
- 15/2 a private descendant of the unit on which one of these with_clauses appear; or
- 16/2 a pragma within a context clause.

16.a/2 **Ramification:** These rules apply only if all of the with_clauses that mention the name include the reserved word **private**. They do not apply if the name is mentioned in any with_clause that does not include **private**.

16.b/2 **Reason:** These rules make the library item visible anywhere that is not visible outside the subsystem rooted at the compilation_unit having the private with_clause, including private parts of packages nested in the visible part, private parts of child packages, the visible part of private children, and context clause pragmas like Elaborate_All.

16.c/2 We considered having the scope of a private with_clause not include the visible part. However, that rule would mean that moving a declaration between the visible part and the private part could change its meaning from one legal interpretation to a different legal interpretation. For example:

16.d/2
```
package A is
    function B return Integer;
end A;
```
16.e/2
```
function B return Integer;
```
16.f/2
```
with A;
private with B;
package C is
    use A;
    V1 : Integer := B; -- (1)
private
    V2 : Integer := B; -- (2)
end C;
```

16.g/2 If we say that library subprogram B is not in scope in the visible part of C, then the B at (1) resolves to A.B, while (2) resolves to library unit B. Simply moving a declaration could silently change its meaning. With the legality rule defined above, the B at (1) is illegal. If the user really meant A.B, they still can say that.

17/2 {*AI95-00217-06*} [A library_item mentioned in a limited_with_clause shall be the implicit declaration of the limited view of a library package, not the declaration of a subprogram, generic unit, generic instance, or a renaming.]

17.a/2 **Proof:** This is redundant because only such implicit declarations are visible in a limited_with_clause. See 10.1.6.

18/2 {*AI95-00217-06*} {*AI95-00412-01*} A limited_with_clause shall not appear on a library_unit_body, subunit, or library_unit_renaming_declaration.

18.a/2 **Reason:** {*AI95-00412-01*} We don't allow a limited_with_clause on a library_unit_renaming_declaration because it would be useless and therefore probably is a mistake. A renaming cannot appear in a limited_with_clause (by the rule prior to this one), and a renaming of a limited view cannot appear in a nonlimited_with_clause (because the name would not be within the scope of a with_clause denoting the package, see 8.5.3). Nor could it be the parent of another unit. That doesn't leave anywhere that the name of such a renaming **could** appear, so we simply make writing it illegal.

19/2 {*AI95-00217-06*} A limited_with_clause that names a library package shall not appear:

20/2
- {*AI95-00217-06*} in the context_clause for the explicit declaration of the named library package;

20.a/2 **Reason:** We have to explicitly disallow

20.b/2
```
limited with P;
package P is ...
```

20.c/2 as we can't depend on the semantic dependence rules to do it for us as with regular withs. This says “named” and not “mentioned” in order that

20.d/2
```
limited private with P.Child;
package P is ...
```

20.e/2 can be used to allow a mutual dependence between the private part of P and the private child P.Child, which occurs in interfacing and other problems. Since the child always semantically depends on the parent, this is the only way such a dependence can be broken.

21/2
- {*AI95-00217-06*} in the same context_clause as, or within the scope of, a nonlimited_with_clause that mentions the same library package; or

21.a/2 **Reason:** Such a limited_with_clause could have no effect, and would be confusing. If it is within the scope of a nonlimited_with_clause, or if such a clause is in the context_clause, the full view is available, which strictly provides more information than the limited view.

22/2
- {*AI95-00217-06*} in the same context_clause as, or within the scope of, a use_clause that names an entity declared within the declarative region of the library package.

22.a/2 **Reason:** This prevents visibility issues, where whether an entity is an incomplete or full view depends on how the name of the entity is written. The limited_with_clause cannot be useful, as we must have the full view available in the parent in order for the use_clause to be legal.

NOTES

23/2 3 {*AI95-00217-06*} A library_item mentioned in a nonlimited_with_clause of a compilation unit is visible within the compilation unit and hence acts just like an ordinary declaration. Thus, within a compilation unit that mentions its declaration, the name of a library package can be given in use_clauses and can be used to form expanded names, a library subprogram can be called, and instances of a generic library unit can be declared. If a child of a parent generic package is mentioned in a nonlimited_with_clause, then the corresponding declaration nested within each visible instance is visible within the compilation unit. Similarly, a library_item mentioned in a limited_with_clause of a compilation unit is visible within the compilation unit and thus can be used to form expanded names.

23.a **Ramification:** The rules given for with_clauses are such that the same effect is obtained whether the name of a library unit is mentioned once or more than once by the applicable with_clauses, or even within a given with_clause.

23.b If a with_clause mentions a library_unit_renaming_declaration, it only “mentions” the prefixes appearing explicitly in the with_clause (and the renamed view itself); the with_clause is not defined to mention the ancestors of the renamed entity. Thus, if X renames Y.Z, then “with X;” does not make the declarations of Y or Z visible. Note that this does not cause the dreaded visibility holes mentioned above.

Examples

24/2
```
{AI95-00433-01} package Office is
end Office;
```

25/2
```
{AI95-00433-01} with Ada.Strings.Unbounded;
package Office.Locations is
   type Location is new Ada.Strings.Unbounded.Unbounded_String;
end Office.Locations;
```

26/2
```
{AI95-00433-01} limited with Office.Departments;  -- types are incomplete
private with Office.Locations;    -- only visible in private part
package Office.Employees is
   type Employee is private;
```

27/2
```
   function Dept_Of(Emp : Employee) return access Departments.Department;
   procedure Assign_Dept(Emp  : in out Employee;
                         Dept : access Departments.Department);
```

28/2
```
   ...
private
   type Employee is
      record
         Dept : access Departments.Department;
         Loc : Locations.Location;
         ...
      end record;
end Office.Employees;
```

29/2
```
limited with Office.Employees;
package Office.Departments is
   type Department is private;
```

30/2
```
   function Manager_Of(Dept : Department) return access Employees.Employee;
   procedure Assign_Manager(Dept : in out Department;
                            Mgr  : access Employees.Employee);
   ...
end Office.Departments;
```

31/2 {*AI95-00433-01*} The limited_with_clause may be used to support mutually dependent abstractions that are split across multiple packages. In this case, an employee is assigned to a department, and a department has a manager who is an employee. If a with_clause with the reserved word **private** appears on one library unit and mentions a second library unit, it provides visibility to the second library unit, but restricts that visibility to the private part and body of the first unit. The compiler checks that no use is made of the second unit in the visible part of the first unit.

Extensions to Ada 83

31.a {*extensions to Ada 83*} The syntax rule for with_clause is modified to allow expanded name notation.

31.b A use_clause in a context_clause may be for a package (or type) nested in a library package.

Wording Changes from Ada 83

31.c The syntax rule for context_clause is modified to more closely reflect the semantics. The Ada 83 syntax rule implies that the use_clauses that appear immediately after a particular with_clause are somehow attached to that with_clause, which is not true. The new syntax allows a use_clause to appear first, but that is prevented by a textual rule that already exists in Ada 83.

31.d The concept of "scope of a with_clause" (which is a region of text) replaces RM83's notion of "apply to" (a with_clause applies to a library_item) The visibility rules are interested in a region of text, not in a set of compilation units.

31.e No need to define "apply to" for use_clauses. Their semantics are fully covered by the "scope (of a use_clause)" definition in 8.4.

Incompatibilities With Ada 95

31.f/2 {*AI95-00220-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** A subprogram body acting as a declaration cannot **with** a private child unit. This would allow public export of types declared in private child packages, and thus cannot be allowed. This was allowed by mistake in Ada 95; a subprogram that does this will now be illegal.

Extensions to Ada 95

31.g/2 {*AI95-00217-06*} {*extensions to Ada 95*} limited_with_clauses are new. They make a limited view of a package visible, where all of the types in the package are incomplete. They facilitate construction of mutually recursive types in multiple packages.

31.h/2 {*AI95-00262-01*} {*extensions to Ada 95*} The syntax rules for with_clause are modified to allow the reserved word **private**. Private with_clauses do not allow the use of their library item in the visible part of their compilation_unit. They also allow using private units in more locations than in Ada 95.

10.1.3 Subunits of Compilation Units

1 [Subunits are like child units, with these (important) differences: subunits support the separate compilation of bodies only (not declarations); the parent contains a body_stub to indicate the existence and place of each of its subunits; declarations appearing in the parent's body can be visible within the subunits.]

Syntax

2 body_stub ::=
subprogram_body_stub | package_body_stub | task_body_stub | protected_body_stub

3/2 {*AI95-00218-03*} subprogram_body_stub ::=
[overriding_indicator]
subprogram_specification **is separate**;

3.a **Discussion:** Although this syntax allows a parent_unit_name, that is disallowed by 10.1.1, “Compilation Units - Library Units”.

4 package_body_stub ::= **package body** defining_identifier **is separate**;

5 task_body_stub ::= **task body** defining_identifier **is separate**;

6 protected_body_stub ::= **protected body** defining_identifier **is separate**;

7 subunit ::= **separate** (parent_unit_name) proper_body

Legality Rules

8/2 {*AI95-00243-01*} {*parent body (of a subunit)*} The *parent body* of a subunit is the body of the program unit denoted by its parent_unit_name. {*subunit*} The term *subunit* is used to refer to a subunit and also to the proper_body of a subunit. The *subunits of a program unit* include any subunit that names that program unit as its parent, as well as any subunit that names such a subunit as its parent (recursively).{*subunit (of a program unit)*}

8.a.1/2 **Reason:** {*AI95-00243-01*} We want any rule that applies to a subunit to apply to a subunit of a subunit as well.

9 The parent body of a subunit shall be present in the current environment, and shall contain a corresponding body_stub with the same defining_identifier as the subunit.

9.a **Discussion:** This can't be a Name Resolution Rule, because a subunit is not a complete context.

10/2 A package_body_stub shall be the completion of a package_declaration or generic_package_-declaration; a task_body_stub shall be the completion of a task declaration; a protected_body_stub shall be the completion of a protected declaration.

11 In contrast, a subprogram_body_stub need not be the completion of a previous declaration, [in which case the _stub declares the subprogram]. If the _stub is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a subprogram_body_stub that completes a declaration shall conform fully to that of the declaration. {*full conformance (required)*}

Discussion: The part about subprogram_body_stubs echoes the corresponding rule for subprogram_bodies in 6.3, “Subprogram Bodies”. 11.a

A subunit that corresponds to a body_stub shall be of the same kind (package_, subprogram_, task_, or protected_) as the body_stub. The profile of a subprogram_body subunit shall be fully conformant to that of the corresponding body_stub. {*full conformance (required)*} 12

A body_stub shall appear immediately within the declarative_part of a compilation unit body. This rule does not apply within an instance of a generic unit. 13

Discussion: {*methodological restriction*} This is a methodological restriction; that is, it is not necessary for the semantics of the language to make sense. 13.a

The defining_identifiers of all body_stubs that appear immediately within a particular declarative_part shall be distinct. 14

Post-Compilation Rules

For each body_stub, there shall be a subunit containing the corresponding proper_body. 15

NOTES

4 The rules in 10.1.4, “The Compilation Process” say that a body_stub is equivalent to the corresponding proper_body. This implies: 16

- Visibility within a subunit is the visibility that would be obtained at the place of the corresponding body_stub (within the parent body) if the context_clause of the subunit were appended to that of the parent body. 17

Ramification: Recursively. Note that this transformation might make the parent illegal; hence it is not a true equivalence, but applies only to visibility within the subunit. 17.a

- The effect of the elaboration of a body_stub is to elaborate the subunit. 18

Ramification: The elaboration of a subunit is part of its parent body's elaboration, whereas the elaboration of a child unit is not part of its parent declaration's elaboration. 18.a

Ramification: A library_item that is mentioned in a with_clause of a subunit can be hidden (from direct visibility) by a declaration (with the same identifier) given in the subunit. Moreover, such a library_item can even be hidden by a declaration given within the parent body since a library unit is declared in its parent's declarative region; this however does not affect the interpretation of the with_clauses themselves, since only library_items are visible or directly visible in with_clauses. 18.b

The body of a protected operation cannot be a subunit. This follows from the syntax rules. The body of a protected unit can be a subunit. 18.c

Examples

The package Parent is first written without subunits: 19

20
```
package Parent is
    procedure Inner;
end Parent;
```

21
```
with Ada.Text_IO;
package body Parent is
    Variable : String := "Hello, there.";
    procedure Inner is
    begin
        Ada.Text_IO.Put_Line(Variable);
    end Inner;
end Parent;
```

The body of procedure Inner may be turned into a subunit by rewriting the package body as follows (with the declaration of Parent remaining the same): 22

23
```
package body Parent is
    Variable : String := "Hello, there.";
    procedure Inner is separate;
end Parent;
```

24
```
with Ada.Text_IO;
separate(Parent)
procedure Inner is
begin
    Ada.Text_IO.Put_Line(Variable);
end Inner;
```

Extensions to Ada 83

24.a {*extensions to Ada 83*} Subunits of the same ancestor library unit are no longer restricted to have distinct identifiers. Instead, we require only that the full expanded names be distinct.

Extensions to Ada 95

24.b/2 {*AI95-00218-03*} {*extensions to Ada 95*} An overriding_indicator (see 8.3.1) is allowed on a subprogram stub.

Wording Changes from Ada 95

24.c/2 {*AI95-00243-01*} Clarified that a subunit of a subunit is still a subunit.

10.1.4 The Compilation Process

1 {*environment*} {*environment declarative_part*} Each compilation unit submitted to the compiler is compiled in the context of an *environment* declarative_part (or simply, an *environment*), which is a conceptual declarative_part that forms the outermost declarative region of the context of any compilation. At run time, an environment forms the declarative_part of the body of the environment task of a partition (see 10.2, "Program Execution").

1.a **Ramification:** At compile time, there is no particular construct that the declarative region is considered to be nested within — the environment is the universe.

1.b **To be honest:** The environment is really just a portion of a declarative_part, since there might, for example, be bodies that do not yet exist.

2 The declarative_items of the environment are library_items appearing in an order such that there are no forward semantic dependences. Each included subunit occurs in place of the corresponding stub. The visibility rules apply as if the environment were the outermost declarative region, except that with_clauses are needed to make declarations of library units visible (see 10.1.2).

3/2 {*AI95-00217-06*} The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined. The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment are implementation defined.

3.a **Implementation defined:** The mechanisms for creating an environment and for adding and replacing compilation units.

3.a.1/2 **Implementation defined:** The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment.

3.b **Ramification:** The traditional model, used by most Ada 83 implementations, is that one places a compilation unit in the environment by compiling it. Other models are possible. For example, an implementation might define the environment to be a directory; that is, the compilation units in the environment are all the compilation units in the source files contained in the directory. In this model, the mechanism for replacing a compilation unit with a new one is simply to edit the source file containing that compilation unit.

Name Resolution Rules

4/1 {*8652/0032*} {*AI95-00192-01*} If a library_unit_body that is a subprogram_body is submitted to the compiler, it is interpreted only as a completion if a library_unit_declaration with the same defining_program_unit_name already exists in the environment for a subprogram other than an instance of a generic subprogram or for a generic subprogram (even if the profile of the body is not type conformant with that of the declaration); otherwise the subprogram_body is interpreted as both the declaration and body of a library subprogram. {*type conformance* [partial]}

Ramification: The principle here is that a subprogram_body should be interpreted as only a completion if and only if it "might" be legal as the completion of some preexisting declaration, where "might" is defined in a way that does not require overload resolution to determine. 4.a

Hence, if the preexisting declaration is a subprogram_declaration or generic_subprogram_declaration, we treat the new subprogram_body as its completion, because it "might" be legal. If it turns out that the profiles don't fully conform, it's an error. In all other cases (the preexisting declaration is a package or a generic package, or an instance of a generic subprogram, or a renaming, or a "spec-less" subprogram, or in the case where there is no preexisting thing), the subprogram_body declares a new subprogram. 4.b

See also AI83-00266/09. 4.c

Legality Rules

When a compilation unit is compiled, all compilation units upon which it depends semantically shall already exist in the environment; {*consistency (among compilation units)*} the set of these compilation units shall be *consistent* in the sense that the new compilation unit shall not semantically depend (directly or indirectly) on two different versions of the same compilation unit, nor on an earlier version of itself. 5

Discussion: For example, if package declarations A and B both say "**with** X;", and the user compiles a compilation unit that says "**with** A, B;", then the A and B have to be talking about the same version of X. 5.a

Ramification: What it means to be a "different version" is not specified by the language. In some implementations, it means that the compilation unit has been recompiled. In others, it means that the source of the compilation unit has been edited in some significant way. 5.b

Note that an implementation cannot require the existence of compilation units upon which the given one does not semantically depend. For example, an implementation is required to be able to compile a compilation unit that says "**with** A;" when A's body does not exist. It has to be able to detect errors without looking at A's body. 5.c

Similarly, the implementation has to be able to compile a call to a subprogram for which a pragma Inline has been specified without seeing the body of that subprogram — inlining would not be achieved in this case, but the call is still legal. 5.d

{*AI95-00217-06*} The second rule applies to limited views as well as the full view of a compilation unit. That means that an implementation needs a way to enforce consistency of limited views, not just of full views. 5.e/2

Implementation Permissions

{*AI95-00217-06*} The implementation may require that a compilation unit be legal before it can be mentioned in a limited_with_clause or it can be inserted into the environment. 6/2

{*AI95-00214-01*} When a compilation unit that declares or renames a library unit is added to the environment, the implementation may remove from the environment any preexisting library_item or subunit with the same full expanded name. When a compilation unit that is a subunit or the body of a library unit is added to the environment, the implementation may remove from the environment any preexisting version of the same compilation unit. When a compilation unit that contains a body_stub is added to the environment, the implementation may remove any preexisting library_item or subunit with the same full expanded name as the body_stub. When a given compilation unit is removed from the environment, the implementation may also remove any compilation unit that depends semantically upon the given one. If the given compilation unit contains the body of a subprogram to which a pragma Inline applies, the implementation may also remove any compilation unit containing a call to that subprogram. 7/2

Ramification: The permissions given in this paragraph correspond to the traditional model, where compilation units enter the environment by being compiled into it, and the compiler checks their legality at that time. A implementation model in which the environment consists of all source files in a given directory might not want to take advantage of these permissions. Compilation units would not be checked for legality as soon as they enter the environment; legality checking would happen later, when compilation units are compiled. In this model, compilation units might never be automatically removed from the environment; they would be removed when the user explicitly deletes a source file. 7.a

Note that the rule is recursive: if the above permission is used to remove a compilation unit containing an inlined subprogram call, then compilation units that depend semantically upon the removed one may also be removed, and so on. 7.b

Note that here we are talking about dependences among existing compilation units in the environment; it doesn't matter what with_clauses are attached to the new compilation unit that triggered all this. 7.c

7.d An implementation may have other modes in which compilation units in addition to the ones mentioned above are removed. For example, an implementation might inline subprogram calls without an explicit pragma Inline. If so, it either has to have a mode in which that optimization is turned off, or it has to automatically regenerate code for the inlined calls without requiring the user to resubmit them to the compiler.

7.d.1/2 **Discussion:** {*8652/0108*} {*AI95-00077-01*} {*AI95-00114-01*} In the standard mode, implementations may only remove units from the environment for one of the reasons listed here, or in response to an explicit user command to modify the environment. It is not intended that the act of compiling a unit is one of the “mechanisms” for removing units other than those specified by this International Standard.

7.e/2 {*AI95-00214-01*} These rules are intended to ensure that an implementation never need keep more than one compilation unit with any full expanded name. In particular, it is not necessary to be able to have a subunit and a child unit with the same name in the environment at one time.

NOTES

8 5 The rules of the language are enforced across compilation and compilation unit boundaries, just as they are enforced within a single compilation unit.

8.a **Ramification:** Note that Section 1 requires an implementation to detect illegal compilation units at compile time.

9 6 {*library*} An implementation may support a concept of a *library*, which contains library_items. If multiple libraries are supported, the implementation has to define how a single environment is constructed when a compilation unit is submitted to the compiler. Naming conflicts between different libraries might be resolved by treating each library as the root of a hierarchy of child library units. {*program library: See library*}

9.a **Implementation Note:** Alternatively, naming conflicts could be resolved via some sort of hiding rule.

9.b **Discussion:** For example, the implementation might support a command to import library Y into library X. If a root library unit called LU (that is, Standard.LU) exists in Y, then from the point of view of library X, it could be called Y.LU. X might contain library units that say, “**with** Y.LU;”.

10 7 A compilation unit containing an instantiation of a separately compiled generic unit does not semantically depend on the body of the generic unit. Therefore, replacing the generic body in the environment does not result in the removal of the compilation unit containing the instantiation.

10.a **Implementation Note:** Therefore, implementations have to be prepared to automatically instantiate generic bodies at link-time, as needed. This might imply a complete automatic recompilation, but it is the intent of the language that generic bodies can be (re)instantiated without forcing all of the compilation units that semantically depend on the compilation unit containing the instantiation to be recompiled.

Extensions to Ada 83

10.b/2 {*AI95-00077-01*} {*AI95-00114-01*} {*extensions to Ada 83*} Ada 83 allowed implementations to require that the body of a generic unit be available when the instantiation is compiled; that permission is dropped in Ada 95. This isn't really an extension (it doesn't allow Ada users to write anything that they couldn't in Ada 83), but there isn't a more appropriate category, and it does allow users more flexibility when developing programs.

Wording Changes from Ada 95

10.c/2 {*8652/0032*} {*AI95-00192-01*} **Corrigendum:** The wording was clarified to ensure that a subprogram_body is not considered a completion of an instance of a generic subprogram.

10.d/2 {*AI95-00214-01*} The permissions to remove a unit from the environment were clarified to ensure that it is never necessary to keep multiple (sub)units with the same full expanded name in the environment.

10.e/2 {*AI95-00217-06*} Units mentioned in a limited_with_clause were added to several rules; limited views have the same presence in the environment as the corresponding full views.

10.1.5 Pragmas and Program Units

1 [This subclause discusses pragmas related to program units, library units, and compilations.]

Name Resolution Rules

2 {*program unit pragma* [distributed]} {*pragma, program unit* [distributed]} Certain pragmas are defined to be *program unit pragmas*. {*apply (to a program unit by a program unit pragma)* [partial]} A name given as the argument of a program unit pragma shall resolve to denote the declarations or renamings of one or more program units that occur immediately within the declarative region or compilation in which the pragma immediately occurs, or it shall resolve to denote the declaration of the immediately enclosing program

unit (if any); the pragma applies to the denoted program unit(s). If there are no names given as arguments, the pragma applies to the immediately enclosing program unit.

> **Ramification:** The fact that this is a Name Resolution Rule means that the pragma will not apply to declarations from outer declarative regions. 2.a

Legality Rules

A program unit pragma shall appear in one of these places: 3

- At the place of a compilation_unit, in which case the pragma shall immediately follow in the same compilation (except for other pragmas) a library_unit_declaration that is a subprogram_-declaration, generic_subprogram_declaration, or generic_instantiation, and the pragma shall have an argument that is a name denoting that declaration. 4

> **Ramification:** The name has to denote the immediately preceding library_unit_declaration. 4.a

- {*8652/0033*} {*AI95-00136-01*} Immediately within the visible part of a program unit and before any nested declaration (but not within a generic formal part), in which case the argument, if any, shall be a direct_name that denotes the immediately enclosing program unit declaration. 5/1

> **Ramification:** The argument is optional in this case. 5.a

- At the place of a declaration other than the first, of a declarative_part or program unit declaration, in which case the pragma shall have an argument, which shall be a direct_name that denotes one or more of the following (and nothing else): a subprogram_declaration, a generic_-subprogram_declaration, or a generic_instantiation, of the same declarative_part or program unit declaration. 6

> **Ramification:** If you want to denote a subprogram_body that is not a completion, or a package_declaration, for example, you have to put the pragma inside. 6.a

{*library unit pragma* [distributed]} {*pragma, library unit* [distributed]} {*program unit pragma (library unit pragmas)* [partial]} {*pragma, program unit (library unit pragmas)* [partial]} Certain program unit pragmas are defined to be *library unit pragmas*. The name, if any, in a library unit pragma shall denote the declaration of a library unit. 7

> **Ramification:** This, together with the rules for program unit pragmas above, implies that if a library unit pragma applies to a subprogram_declaration (and similar things), it has to appear immediately after the compilation_unit, whereas if the pragma applies to a package_declaration, a subprogram_body that is not a completion (and similar things), it has to appear inside, as the first declarative_item. 7.a

Static Semantics

{*8652/0034*} {*AI95-00041-01*} A library unit pragma that applies to a generic unit does not apply to its instances, unless a specific rule for the pragma specifies the contrary. 7.1/1

Post-Compilation Rules

{*configuration pragma* [distributed]} {*pragma, configuration* [distributed]} Certain pragmas are defined to be *configuration pragmas*; they shall appear before the first compilation_unit of a compilation. [They are generally used to select a partition-wide or system-wide option.] The pragma applies to all compilation_units appearing in the compilation, unless there are none, in which case it applies to all future compilation_units compiled into the same environment. 8

Implementation Permissions

{*AI95-00212-01*} An implementation may require that configuration pragmas that select partition-wide or system-wide options be compiled when the environment contains no library_items other than those of the predefined environment. In this case, the implementation shall still accept configuration pragmas in individual compilations that confirm the initially selected partition-wide or system-wide options. 9/2

Implementation Advice

10/1 {*8652/0034*} {*AI95-00041-01*} When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance.

10.a/2 **Implementation Advice:** When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance.

Wording Changes from Ada 95

10.b/2 {*8652/0033*} {*AI95-00136-01*} **Corrigendum:** The wording was corrected to ensure that a program unit pragma cannot appear in private parts or generic formal parts.

10.c/2 {*8652/0034*} {*AI95-00041-01*} **Corrigendum:** The wording was clarified to explain the meaning of program unit and library unit pragmas in generic units.

10.d/2 The Implementation Advice added by the Corrigendum was moved, as it was not in the normal order. (This changes the paragraph number.) It originally was directly after the new Static Semantics rule.

10.e/2 {*AI95-00212-01*} The permission to place restrictions was clarified to:

10.f/2
- Ensure that it applies only to partition-wide configuration pragmas, not ones like Assertion_Policy (see 11.4.2), which can be different in different units; and

10.g/2
- Ensure that confirming pragmas are always allowed.

10.1.6 Environment-Level Visibility Rules

1 [The normal visibility rules do not apply within a parent_unit_name or a context_clause, nor within a pragma that appears at the place of a compilation unit. The special visibility rules for those contexts are given here.]

Static Semantics

2/2 {*AI95-00217-06*} {*AI95-00312-01*} {*directly visible (within the parent_unit_name of a library unit)* [partial]} {*visible (within the parent_unit_name of a library unit)* [partial]} {*directly visible (within a with_clause)* [partial]} {*visible (within a with_clause)* [partial]} Within the parent_unit_name at the beginning of an explicit library_item, and within a nonlimited_with_clause, the only declarations that are visible are those that are explicit library_items of the environment, and the only declarations that are directly visible are those that are explicit root library_items of the environment. Within a limited_with_clause, the only declarations that are visible are those that are the implicit declaration of the limited view of a library package of the environment, and the only declarations that are directly visible are those that are the implicit declaration of the limited view of a root library package.

2.a **Ramification:** In "**package** P.Q.R **is** ... **end** P.Q.R;", this rule requires P to be a root library unit, and Q to be a library unit (because those are the things that are directly visible and visible). Note that visibility does not apply between the "**end**" and the ";".

2.b Physically nested declarations are not visible at these places.

2.c Although Standard is visible at these places, it is impossible to name it, since it is not directly visible, and it has no parent.

2.c.1/2 {*AI95-00217-06*} Only compilation units defining limited views can be mentioned in a limited_with_clause, while only compilation units defining full views (that is, the explicit declarations) can be mentioned in a nonlimited_with_clause. This resolves the conflict inherent in having two compilation units with the same defining name.

2.d/2 *This paragraph was deleted.*{*AI95-00312-01*}

3 {*directly visible (within a use_clause in a context_clause)* [partial]} {*visible (within a use_clause in a context_clause)* [partial]} {*directly visible (within a pragma in a context_clause)* [partial]} {*visible (within a pragma in a context_clause)* [partial]} Within a use_clause or pragma that is within a context_clause, each library_item mentioned in a previous with_clause of the same context_clause is visible, and each

root library_item so mentioned is directly visible. In addition, within such a use_clause, if a given declaration is visible or directly visible, each declaration that occurs immediately within the given declaration's visible part is also visible. No other declarations are visible or directly visible.

Discussion: Note the word “same”. For example, if a with_clause on a declaration mentions X, this does not make X visible in use_clauses and pragmas that are on the body. The reason for this rule is the one-pass context_clauses Language Design Principle. 3.a

Note that the second part of the rule does not mention pragmas. 3.b

4 {*directly visible (within the parent_unit_name of a subunit)* [partial]} {*visible (within the parent_unit_name of a subunit)* [partial]} Within the parent_unit_name of a subunit, library_items are visible as they are in the parent_unit_name of a library_item; in addition, the declaration corresponding to each body_stub in the environment is also visible.

Ramification: For a subprogram without a separate subprogram_declaration, the body_stub itself is the declaration. 4.a

5 {*directly visible (within a pragma that appears at the place of a compilation unit)* [partial]} {*visible (within a pragma that appears at the place of a compilation unit)* [partial]} Within a pragma that appears at the place of a compilation unit, the immediately preceding library_item and each of its ancestors is visible. The ancestor root library_item is directly visible.

6/2 {*AI95-00312-01*} {*notwithstanding*} Notwithstanding the rules of 4.1.3, an expanded name in a with_clause, a pragma in a context_clause, or a pragma that appears at the place of a compilation unit may consist of a prefix that denotes a generic package and a selector_name that denotes a child of that generic package. [(The child is necessarily a generic unit; see 10.1.1.)]

Reason: This rule allows **with** A.B; and **pragma** Elaborate(A.B); where A is a generic library package and B is one of its (generic) children. This is necessary because it is not normally legal to use an expanded name to reach inside a generic package. 6.a/2

Wording Changes from Ada 83

The special visibility rules that apply within a parent_unit_name or a context_clause, and within a pragma that appears at the place of a compilation_unit are clarified. 6.b

Note that a context_clause is not part of any declarative region. 6.c

We considered making the visibility rules within parent_unit_names and context_clauses follow from the context of compilation. However, this attempt failed for various reasons. For example, it would require use_clauses in context_clauses to be within the declarative region of Standard, which sounds suspiciously like a kludge. And we would still need a special rule to prevent seeing things (in our own context_clause) that were with-ed by our parent, etc. 6.d

Wording Changes from Ada 95

{*AI95-00217-06*} Added separate visibility rules for limited_with_clauses; the existing rules apply only to nonlimited_with_clauses. 6.e/2

{*AI95-00312-01*} Clarified that the name of a generic child unit may appear in a pragma in a context_clause. 6.f/2

10.2 Program Execution

1 {*program*} {*program execution*} {*running a program: See program execution*} An Ada *program* consists of a set of *partitions*[, which can execute in parallel with one another, possibly in a separate address space, and possibly on a separate computer.]

Post-Compilation Rules

2 {*partition* [distributed]} {*partition building*} A partition is a program or part of a program that can be invoked from outside the Ada implementation. [For example, on many systems, a partition might be an executable file generated by the system linker.] {*explicitly assign*} The user can *explicitly assign* library units to a partition. The assignment is done in an implementation-defined manner. The compilation units included in a partition are those of the explicitly assigned library units, as well as other compilation units *needed by* those library units. The compilation units needed by a given compilation unit are determined as follows (unless specified otherwise via an implementation-defined pragma, or by some other implementation-defined means): {*linking: See partition building*} {*compilation units needed (by a compilation unit)* [distributed]} {*needed (of a compilation unit by another)* [distributed]}

2.a **Discussion:** From a run-time point of view, an Ada 95 partition is identical to an Ada 83 program — implementations were always allowed to provide inter-program communication mechanisms. The additional semantics of partitions is that interfaces between them can be defined to obey normal language rules (as is done in Annex E, “Distributed Systems”), whereas interfaces between separate programs had no particular semantics.

2.b **Implementation defined:** The manner of explicitly assigning library units to a partition.

2.c **Implementation defined:** The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit.

2.d **Discussion:** There are no pragmas that “specify otherwise” defined by the core language. However, an implementation is allowed to provide such pragmas, and in fact Annex E, “Distributed Systems” defines some pragmas whose semantics includes reducing the set of compilation units described here.

3 • A compilation unit needs itself;

4 • If a compilation unit is needed, then so are any compilation units upon which it depends semantically;

5 • If a library_unit_declaration is needed, then so is any corresponding library_unit_body;

6/2 • {*AI95-00217-06*} If a compilation unit with stubs is needed, then so are any corresponding subunits;

6.a **Discussion:** Note that in the environment, the stubs are replaced with the corresponding proper_bodies.

6.1/2 • {*AI95-00217-06*} If the (implicit) declaration of the limited view of a library package is needed, then so is the explicit declaration of the library package.

6.b **Discussion:** Note that a child unit is not included just because its parent is included — to include a child, mention it in a with_clause.

6.c/2 {*AI95-00217-06*} A package is included in a partition even if the only reference to it is in a limited_with_clause. While this isn't strictly necessary (no objects of types imported from such a unit can be created), it ensures that all incomplete types are eventually completed, and is the least surprising option.

7 {*main subprogram (for a partition)*} The user can optionally designate (in an implementation-defined manner) one subprogram as the *main subprogram* for the partition. A main subprogram, if specified, shall be a subprogram.

7.a **Discussion:** This may seem superfluous, since it follows from the definition. But we would like to have every error message that might be generated (before run time) by an implementation correspond to some explicitly stated “shall” rule.

7.b Of course, this does not mean that the “shall” rules correspond one-to-one with an implementation's error messages. For example, the rule that says overload resolution “shall” succeed in producing a single interpretation would correspond to many error messages in a good implementation — the implementation would want to explain to the user exactly why overload resolution failed. This is especially true for the syntax rules — they are considered part of overload resolution, but in most cases, one would expect an error message based on the particular syntax rule that was violated.

Implementation defined: The manner of designating the main subprogram of a partition. 7.c

Ramification: An implementation cannot require the user to specify, say, all of the library units to be included. It has to support, for example, perhaps the most typical case, where the user specifies just one library unit, the main program. The implementation has to do the work of tracking down all the other ones. 7.d

{*environment task*} Each partition has an anonymous *environment task*[, which is an implicit outermost task whose execution elaborates the library_items of the environment declarative_part, and then calls the main subprogram, if there is one. A partition's execution is that of its tasks.] 8

Ramification: An environment task has no master; all nonenvironment tasks have masters. 8.a

An implementation is allowed to support multiple concurrent executions of the same partition. 8.b

[The order of elaboration of library units is determined primarily by the *elaboration dependences*.] {*elaboration dependence (library_item on another)*} {*dependence (elaboration)*} There is an elaboration dependence of a given library_item upon another if the given library_item or any of its subunits depends semantically on the other library_item. In addition, if a given library_item or any of its subunits has a pragma Elaborate or Elaborate_All that names another library unit, then there is an elaboration dependence of the given library_item upon the body of the other library unit, and, for Elaborate_All only, upon each library_item needed by the declaration of the other library unit. 9

Discussion: {*8652/0107*} {*AI95-00180-01*} {*AI95-00256-01*} “Mentions” was used informally in the above rule; it was not intended to refer to the definition of *mentions* in 10.1.2. It was changed to “names” to make this clear. 9.a.1/2

See above for a definition of which library_items are “needed by” a given declaration. 9.a

Note that elaboration dependences are among library_items, whereas the other two forms of dependence are among compilation units. Note that elaboration dependence includes semantic dependence. It's a little bit sad that pragma Elaborate_Body can't be folded into this mechanism. It follows from the definition that the elaboration dependence relationship is transitive. Note that the wording of the rule does not need to take into account a semantic dependence of a library_item or one of its subunits upon a subunit of a different library unit, because that can never happen. 9.b

The environment task for a partition has the following structure: 10

11
```
task Environment_Task;
```

12/2
```
task body Environment_Task is
    ... (1) -- The environment declarative_part
            -- (that is, the sequence of library_items) goes here.
begin
    ... (2) -- Call the main subprogram, if there is one.
end Environment_Task;
```

Ramification: The name of the environment task is written in italics here to indicate that this task is anonymous. 12.a

Discussion: The model is different for a “passive partition” (see E.1). Either there is no environment task, or its sequence_of_statements is an infinite loop rather than a call on a main subprogram. 12.b

{*environment declarative_part (for the environment task of a partition)* [partial]} The environment declarative_part at (1) is a sequence of declarative_items consisting of copies of the library_items included in the partition[. The order of elaboration of library_items is the order in which they appear in the environment declarative_part]: 13

- The order of all included library_items is such that there are no forward elaboration dependences. 14

Ramification: This rule is written so that if a library_item depends on itself, we don't require it to be elaborated before itself. See AI83-00113/12. This can happen only in pathological circumstances. For example, if a library subprogram_body has no corresponding subprogram_declaration, and one of the subunits of the subprogram_body mentions the subprogram_body in a with_clause, the subprogram_body will depend on itself. For another example, if a library_unit_body applies a pragma Elaborate_All to its own declaration, then the library_unit_body will depend on itself. 14.a

15 • Any included library_unit_declaration to which a pragma Elaborate_Body applies is immediately followed by its library_unit_body, if included.

15.a **Discussion:** This implies that the body of such a library unit shall not "with" any of its own children, or anything else that depends semantically upon the declaration of the library unit.

16 • All library_items declared pure occur before any that are not declared pure.

17 • All preelaborated library_items occur before any that are not preelaborated.

17.a **Discussion:** Normally, if two partitions contain the same compilation unit, they each contain a separate *copy* of that compilation unit. See Annex E, "Distributed Systems" for cases where two partitions share the same copy of something.

17.b There is no requirement that the main subprogram be elaborated last. In fact, it is possible to write a partition in which the main subprogram cannot be elaborated last.

17.c **Ramification:** This declarative_part has the properties required of all environments (see 10.1.4). However, the environment declarative_part of a partition will typically contain fewer compilation units than the environment declarative_part used at compile time — only the "needed" ones are included in the partition.

18 There shall be a total order of the library_items that obeys the above rules. The order is otherwise implementation defined.

18.a **Discussion:** The only way to violate this rule is to have Elaborate, Elaborate_All, or Elaborate_Body pragmas that cause circular ordering requirements, thus preventing an order that has no forward elaboration dependences.

18.b **Implementation defined:** The order of elaboration of library_items.

18.c **To be honest:** {*requires a completion (library_unit_declaration)* [partial]} {*notwithstanding*} Notwithstanding what the RM95 says elsewhere, each rule that requires a declaration to have a corresponding completion is considered to be a Post-Compilation Rule when the declaration is that of a library unit.

18.d **Discussion:** Such rules may be checked at "link time," for example. Rules requiring the completion to have certain properties, on the other hand, are checked at compile time of the completion.

19 The full expanded names of the library units and subunits included in a given partition shall be distinct.

19.a **Reason:** This is a Post-Compilation Rule because making it a Legality Rule would violate the Language Design Principle labeled "legality determinable via semantic dependences."

20 The sequence_of_statements of the environment task (see (2) above) consists of either:

21 • A call to the main subprogram, if the partition has one. If the main subprogram has parameters, they are passed; where the actuals come from is implementation defined. What happens to the result of a main function is also implementation defined.

21.a **Implementation defined:** Parameter passing and function return for the main subprogram.

22 or:

23 • A null_statement, if there is no main subprogram.

23.a **Discussion:** For a passive partition, either there is no environment task, or its sequence_of_statements is an infinite loop. See E.1.

24 The mechanisms for building and running partitions are implementation defined. [These might be combined into one operation, as, for example, in dynamic linking, or "load-and-go" systems.]

24.a **Implementation defined:** The mechanisms for building and running partitions.

Dynamic Semantics

25 {*execution (program)* [partial]} The execution of a program consists of the execution of a set of partitions. Further details are implementation defined. {*execution (partition)* [partial]} The execution of a partition starts with the execution of its environment task, ends when the environment task terminates, and includes the executions of all tasks of the partition. [The execution of the (implicit) task_body of the environment task acts as a master for all other tasks created as part of the execution of the partition. When the

environment task completes (normally or abnormally), it waits for the termination of all such tasks, and then finalizes any remaining objects of the partition.]

Ramification: The "further details" mentioned above include, for example, program termination — it is implementation defined. There is no need to define it here; it's entirely up to the implementation whether it wants to consider the program as a whole to exist beyond the existence of individual partitions. 25.a

Implementation defined: The details of program execution, including program termination. 25.b

To be honest: {*termination (of a partition)* [partial]} {*normal termination (of a partition)* [partial]} {*termination (normal)* [partial]} {*abnormal termination (of a partition)* [partial]} {*termination (abnormal)* [partial]} The execution of the partition terminates (normally or abnormally) when the environment task terminates (normally or abnormally, respectively). 25.c

Bounded (Run-Time) Errors

{*bounded error (cause)* [partial]} {*Program_Error (raised by failure of run-time check)*} Once the environment task has awaited the termination of all other tasks of the partition, any further attempt to create a task (during finalization) is a bounded error, and may result in the raising of Program_Error either upon creation or activation of the task. {*unspecified* [partial]} If such a task is activated, it is not specified whether the task is awaited prior to termination of the environment task. 26

Implementation Requirements

The implementation shall ensure that all compilation units included in a partition are consistent with one another, and are legal according to the rules of the language. 27

Discussion: The consistency requirement implies that a partition cannot contain two versions of the same compilation unit. That is, a partition cannot contain two different library units with the same full expanded name, nor two different bodies for the same program unit. For example, suppose we compile the following: 27.a

27.b
```
package A is -- Version 1.
    ...
end A;
```

27.c
```
with A;
package B is
end B;
```

27.d
```
package A is -- Version 2.
    ...
end A;
```

27.e
```
with A;
package C is
end C;
```

It would be wrong for a partition containing B and C to contain both versions of A. Typically, the implementation would require the use of Version 2 of A, which might require the recompilation of B. Alternatively, the implementation might automatically recompile B when the partition is built. A third alternative would be an incremental compiler that, when Version 2 of A is compiled, automatically patches the object code for B to reflect the changes to A (if there are any relevant changes — there might not be any). 27.f

An implementation that supported fancy version management might allow the use of Version 1 in some circumstances. In no case can the implementation allow the use of both versions in the same partition (unless, of course, it can prove that the two versions are semantically identical). 27.g

The core language says nothing about inter-partition consistency; see also Annex E, "Distributed Systems". 27.h

Implementation Permissions

{*active partition*} The kind of partition described in this clause is known as an *active* partition. An implementation is allowed to support other kinds of partitions, with implementation-defined semantics. 28

Implementation defined: The semantics of any nonactive partitions supported by the implementation. 28.a

Discussion: Annex E, "Distributed Systems" defines the concept of passive partitions; they may be thought of as a partition without an environment task, or as one with a particularly simple form of environment task, having an infinite loop rather than a call on a main subprogram as its sequence_of_statements. 28.b

29 An implementation may restrict the kinds of subprograms it supports as main subprograms. However, an implementation is required to support all main subprograms that are public parameterless library procedures.

29.a **Ramification:** The implementation is required to support main subprograms that are procedures declared by generic_instantiations, as well as those that are children of library units other than Standard. Generic units are, of course, not allowed to be main subprograms, since they are not subprograms.

29.b Note that renamings are irrelevant to this rule. This rules says which subprograms (not views) have to be supported. The implementation can choose any way it wants for the user to indicate which subprogram should be the main subprogram. An implementation might allow any name of any view, including those declared by renamings. Another implementation might require it to be the original name. Another implementation still might use the name of the source file or some such thing.

30 If the environment task completes abnormally, the implementation may abort any dependent tasks.

30.a **Reason:** If the implementation does not take advantage of this permission, the normal action takes place — the environment task awaits those tasks.

30.b The possibility of aborting them is not shown in the *Environment_Task* code above, because there is nowhere to put an exception_handler that can handle exceptions raised in both the environment declarative_part and the main subprogram, such that the dependent tasks can be aborted. If we put an exception_handler in the body of the environment task, then it won't handle exceptions that occur during elaboration of the environment declarative_part. If we were to move those things into a nested block_statement, with the exception_handler outside that, then the block_statement would await the library tasks we are trying to abort.

30.c Furthermore, this is merely a permission, and is not fundamental to the model, so it is probably better to state it separately anyway.

30.d Note that implementations (and tools like debuggers) can have modes that provide other behaviors in addition.

NOTES

31 8 An implementation may provide inter-partition communication mechanism(s) via special packages and pragmas. Standard pragmas for distribution and methods for specifying inter-partition communication are defined in Annex E, “Distributed Systems”. If no such mechanisms are provided, then each partition is isolated from all others, and behaves as a program in and of itself.

31.a **Ramification:** Not providing such mechanisms is equivalent to disallowing multi-partition programs.

31.b An implementation may provide mechanisms to facilitate checking the consistency of library units elaborated in different partitions; Annex E, “Distributed Systems” does so.

32 9 Partitions are not required to run in separate address spaces. For example, an implementation might support dynamic linking via the partition concept.

33 10 An order of elaboration of library_items that is consistent with the partial ordering defined above does not always ensure that each library_unit_body is elaborated before any other compilation unit whose elaboration necessitates that the library_unit_body be already elaborated. (In particular, there is no requirement that the body of a library unit be elaborated as soon as possible after the library_unit_declaration is elaborated, unless the pragmas in subclause 10.2.1 are used.)

34 11 A partition (active or otherwise) need not have a main subprogram. In such a case, all the work done by the partition would be done by elaboration of various library_items, and by tasks created by that elaboration. Passive partitions, which cannot have main subprograms, are defined in Annex E, “Distributed Systems”.

34.a **Ramification:** The environment task is the outermost semantic level defined by the language.

34.b Standard has no private part. This prevents strange implementation-dependences involving private children of Standard having visibility upon Standard's private part. It doesn't matter where the body of Standard appears in the environment, since it doesn't do anything. See Annex A, “Predefined Language Environment”.

34.c Note that elaboration dependence is carefully defined in such a way that if (say) the body of something doesn't exist yet, then there is no elaboration dependence upon the nonexistent body. (This follows from the fact that “needed by” is defined that way, and the elaboration dependences caused by a pragma Elaborate or Elaborate_All are defined in terms of “needed by”.) This property allows us to use the environment concept both at compile time and at partition-construction time/run time.

Extensions to Ada 83

34.d {*extensions to Ada 83*} The concept of partitions is new to Ada 95.

34.e A main subprogram is now optional. The language-defined restrictions on main subprograms are relaxed.

Wording Changes from Ada 83

34.f Ada 95 uses the term “main subprogram” instead of Ada 83's “main program” (which was inherited from Pascal). This is done to avoid confusion — a main subprogram is a subprogram, not a program. The program as a whole is an entirely different thing.

Wording Changes from Ada 95

34.g/2 {*AI95-00256-01*} The mistaken use of “mentions” in the elaboration dependence rule was fixed.

34.h/2 {*AI95-00217-06*} The *needs* relationship was extended to include limited views.

10.2.1 Elaboration Control

1 [{*elaboration control*} This subclause defines pragmas that help control the elaboration order of library_items.]

Language Design Principles

1.a The rules governing preelaboration are designed to allow it to be done largely by bulk initialization of statically allocated storage from information in a “load module” created by a linker. Some implementations may require run-time code to be executed in some cases, but we consider these cases rare enough that we need not further complicate the rules.

1.b It is important that programs be able to declare data structures that are link-time initialized with aggregates, string_literals, and concatenations thereof. It is important to be able to write link-time evaluated expressions involving the First, Last, and Length attributes of such data structures (including variables), because they might be initialized with positional aggregates or string_literals, and we don't want the user to have to count the elements. There is no corresponding need for accessing discriminants, since they can be initialized with a static constant, and then the constant can be referred to elsewhere. It is important to allow link-time initialized data structures involving discriminant-dependent components. It is important to be able to write link-time evaluated expressions involving pointers (both access values and addresses) to the above-mentioned data structures.

1.c The rules also ensure that no Elaboration_Check need be performed for calls on library-level subprograms declared within a preelaborated package. This is true also of the Elaboration_Check on task activation for library level task types declared in a preelaborated package. However, it is not true of the Elaboration_Check on instantiations.

1.d A static expression should never prevent a library unit from being preelaborable.

Syntax

2 The form of a pragma Preelaborate is as follows:

3 **pragma** Preelaborate[(*library_unit_*name)];

4 {*library unit pragma (Preelaborate)* [partial]} {*pragma, library unit (Preelaborate)* [partial]} A pragma Preelaborate is a library unit pragma.

4.1/2 {*AI95-00161-01*} The form of a pragma Preelaborable_Initialization is as follows:

4.2/2 **pragma** Preelaborable_Initialization(direct_name);

Legality Rules

5 {*preelaborable (of an elaborable construct)* [distributed]} An elaborable construct is preelaborable unless its elaboration performs any of the following actions:

5.a **Ramification:** A *preelaborable* construct can be elaborated without using any information that is available only at run time. Note that we don't try to prevent exceptions in preelaborable constructs; if the implementation wishes to generate code to raise an exception, that's OK.

5.b Because there is no flow of control and there are no calls (other than to predefined subprograms), these run-time properties can actually be detected at compile time. This is necessary in order to require compile-time enforcement of the rules.

- 6 The execution of a statement other than a null_statement.

6.a **Ramification:** A preelaborable construct can contain labels and null_statements.

- 7 A call to a subprogram other than a static function.

8 • The evaluation of a primary that is a name of an object, unless the name is a static expression, or statically denotes a discriminant of an enclosing type.

8.a **Ramification:** One can evaluate such a name, but not as a primary. For example, one can evaluate an attribute of the object. One can evaluate an attribute_reference, so long as it does not denote an object, and its prefix does not disobey any of these rules. For example, Obj'Access, Obj'Unchecked_Access, and Obj'Address are generally legal in preelaborated library units.

9/2 • {*AI95-00161-01*} The creation of an object [(including a component)] of a type that does not have preelaborable initialization. Similarly, the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type.

9.a **Ramification:** One can declare these kinds of types, but one cannot create objects of those types.

9.b It is also non-preelaborable to create an object if that will cause the evaluation of a default expression that will call a user-defined function. This follows from the rule above forbidding non-null statements.

9.c/2 *This paragraph was deleted.*{*AI95-00161-01*}

10/2 {*AI95-00403-01*} A generic body is preelaborable only if elaboration of a corresponding instance body would not perform any such actions, presuming that: {*generic contract issue*}

10.1/2 • {*AI95-00403-01*} the actual for each formal private type (or extension) declared within the formal part of the generic unit is a private type (or extension) that does not have preelaborable initialization;

10.2/2 • {*AI95-00403-01*} the actual for each formal type is nonstatic;

10.3/2 • {*AI95-00403-01*} the actual for each formal object is nonstatic; and

10.4/2 • {*AI95-00403-01*} the actual for each formal subprogram is a user-defined subprogram.

10.a.1/2 **Discussion:** {*AI95-00403-01*} This is an “assume-the-worst” rule. The elaboration of a generic unit doesn't perform any of the actions listed above, because its sole effect is to establish that the generic can from now on be instantiated. So the elaboration of the generic itself is not the interesting part when it comes to preelaboration rules. The interesting part is what happens when you elaborate “any instantiation” of the generic. For instance, declaring an object of a limited formal private type might well start tasks, call functions, and do all sorts of non-preelaborable things. We prevent these situations by assuming that the actual parameters are as badly behaved as possible.

10.a **Reason:** Without this rule about generics, we would have to forbid instantiations in preelaborated library units, which would significantly reduce their usefulness.

11/1 {*8652/0035*} {*AI95-00002-01*} {*preelaborated* [partial]} If a pragma Preelaborate (or pragma Pure — see below) applies to a library unit, then it is *preelaborated*. [{*preelaborated* [distributed]} If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all non-preelaborated library_items of the partition.] The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. In addition, all compilation units of a preelaborated library unit shall depend semantically only on compilation units of other preelaborated library units.

11.a **Ramification:** In a generic body, we assume the worst about formal private types and extensions.

11.a.1/1 {*8652/0035*} {*AI95-00002-01*} Subunits of a preelaborated subprogram unit do not need to be preelaborable. This is needed in order to be consistent with units nested in a subprogram body, which do not need to be preelaborable even if the subprogram is preelaborated. However, such subunits cannot depend semantically on non-preelaborated units, which is also consistent with nested units.

11.1/2 {*AI95-00161-01*} {*preelaborable initialization*} The following rules specify which entities have *preelaborable initialization*:

11.2/2 • The partial view of a private type or private extension, a protected type without entry_declarations, a generic formal private type, or a generic formal derived type, have preelaborable initialization if and only if the pragma Preelaborable_Initialization has been applied to them. [A protected type with entry_declarations or a task type never has preelaborable initialization.]

- A component (including a discriminant) of a record or protected type has preelaborable initialization if its declaration includes a default_expression whose execution does not perform any actions prohibited in preelaborable constructs as described above, or if its declaration does not include a default expression and its type has preelaborable initialization. 11.3/2
- A derived type has preelaborable initialization if its parent type has preelaborable initialization and (in the case of a derived record extension) if the non-inherited components all have preelaborable initialization. However, a user-defined controlled type with an overriding Initialize procedure does not have preelaborable initialization. 11.4/2
- {*AI95-00161-01*} {*AI95-00345-01*} A view of a type has preelaborable initialization if it is an elementary type, an array type whose component type has preelaborable initialization, a record type whose components all have preelaborable initialization, or an interface type. 11.5/2

{*AI95-00161-01*} A pragma Preelaborable_Initialization specifies that a type has preelaborable initialization. This pragma shall appear in the visible part of a package or generic package. 11.6/2

{*AI95-00161-01*} {*AI95-00345-01*} If the pragma appears in the first list of basic_declarative_items of a package_specification, then the direct_name shall denote the first subtype of a private type, private extension, or protected type that is not an interface type and is without entry_declarations, and the type shall be declared immediately within the same package as the pragma. If the pragma is applied to a private type or a private extension, the full view of the type shall have preelaborable initialization. If the pragma is applied to a protected type, each component of the protected type shall have preelaborable initialization. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit. 11.7/2

{*AI95-00161-01*} If the pragma appears in a generic_formal_part, then the direct_name shall denote a generic formal private type or a generic formal derived type declared in the same generic_formal_part as the pragma. In a generic_instantiation the corresponding actual type shall have preelaborable initialization. 11.8/2

> **Ramification:** Not only do protected types with entry_declarations and task types not have preelaborable initialization, but they cannot have pragma Preelaborable_Initialization applied to them. 11.b/2

Implementation Advice

In an implementation, a type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package, whether the elaborations occur in distinct executions of the same program, or in executions of distinct programs or partitions that include the given version. 12

> **Implementation Advice:** A type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package. 12.a/2

Syntax

The form of a pragma Pure is as follows: 13

pragma Pure[(*library_unit_*name)]; 14

{*library unit pragma (Pure)* [partial]} {*pragma, library unit (Pure)* [partial]} A pragma Pure is a library unit pragma. 15

Static Semantics

{*AI95-00366-01*} {*pure*} A *pure* library_item is a preelaborable library_item whose elaboration does not perform any of the following actions: 15.1/2

- the elaboration of a variable declaration; 15.2/2

15.3/2 • the evaluation of an allocator of an access-to-variable type; for the purposes of this rule, the partial view of a type is presumed to have non-visible components whose default initialization evaluates such an allocator;

15.a/2 **Reason:** This rule is needed because aggregates can specify the default initialization of a private type or extension using <> or the ancestor subtype of an extension aggregate. The subtype of a component could use an allocator to initialize an access discriminant. Ada 95 did not allow such private types to have preelaborable initialization, so they could not have occurred. Thus this rule is not incompatible with Ada 95.

15.4/2 • the elaboration of the declaration of a named access-to-variable type unless the Storage_Size of the type has been specified by a static expression with value zero or is defined by the language to be zero;

15.b/2 **Discussion:** A remote access-to-class-wide type (see E.2.2) has its Storage_Size defined to be zero.

15.c/2 **Reason:** {*AI95-00366-01*} We disallow most named access-to-object types because an allocator has a side effect; the pool constitutes variable data. We allow access-to-subprogram types because they don't have allocators. We even allow named access-to-object types if they have an empty predefined pool (they can't have a user-defined pool as System.Storage_Pools is not pure). In this case, most attempts to use an allocator are illegal, and any others (in a generic body) will raise Storage_Error.

15.5/2 • the elaboration of the declaration of a named access-to-constant type for which the Storage_Size has been specified by an expression other than a static expression with value zero.

15.d/2 **Discussion:** We allow access-to-constant types so long as there is no user-specified non-zero Storage_Size; if there were a user-specified non-zero Storage_Size restricting the size of the storage pool, allocators would be problematic since the package is supposedly 'stateless', and the allocated size count for the storage pool would represent state.

15.6/2 {*AI95-00366-01*} The Storage_Size for an anonymous access-to-variable type declared at library level in a library unit that is declared pure is defined to be zero.

15.e/2 **Ramification:** This makes allocators illegal for such types (see 4.8), making a storage pool unnecessary for these types. A storage pool would represent state.

15.f/2 Note that access discriminants and access parameters are never library-level, even when they are declared in a type or subprogram declared at library-level. That's because they have their own special accessibility rules (see 3.10.2).

Legality Rules

16/2 *This paragraph was deleted.*{*AI95-00366-01*}

17/2 {*AI95-00366-01*} {*declared pure*} A pragma Pure is used to declare that a library unit is pure. If a pragma Pure applies to a library unit, then its compilation units shall be pure, and they shall depend semantically only on compilation units of other library units that are declared pure. Furthermore, the full view of any partial view declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see 13.13.2).

17.a **To be honest:** A *declared-pure* library unit is one to which a pragma Pure applies. Its declaration and body are also said to be declared pure.

17.b **Discussion:** A declared-pure package is useful for defining types to be shared between partitions with no common address space.

17.c **Reason:** Note that generic packages are not mentioned in the list of things that can contain variable declarations. Note that the Ada 95 rules for deferred constants make them allowable in library units that are declared pure; that isn't true of Ada 83's deferred constants.

17.d/2 **Ramification:** {*AI95-00366-01*} Anonymous access types are allowed.

17.e/2 **Reason:** {*AI95-00366-01*} Ada 95 didn't allow any access types as these (including access-to-subprogram types) cause trouble for Annex E, "Distributed Systems", because such types allow access values in a shared passive partition to designate objects in an active partition, thus allowing inter-address space references. We decided to disallow such uses in the relatively rare cases where they cause problems, rather than making life harder for the majority of users. Types declared in a pure package can be used in remote operations only if they are externally streamable. That simply means that there is a means to transport values of the type; that's automatically true for nonlimited types that don't have an access part. The only tricky part about this is to avoid privacy leakage; that was handled by ensuring that any private types (and private extensions) declared in a pure package that have available stream attributes (which include all nonlimited types by definition) have to be externally streamable.

Implementation Permissions

18/2 {*AI95-00366-01*} If a library unit is declared pure, then the implementation is permitted to omit a call on a library-level subprogram of the library unit if the results are not needed after the call. In addition, the implementation may omit a call on such a subprogram and simply reuse the results produced by an earlier call on the same subprogram, provided that none of the parameters nor any object accessible via access values from the parameters are of a limited type, and the addresses and values of all by-reference actual parameters, the values of all by-copy-in actual parameters, and the values of all objects accessible via access values from the parameters, are the same as they were at the earlier call. [This permission applies even if the subprogram produces other side effects when called.]

18.a/2 **Discussion:** {*AI95-00366-01*} A declared-pure library_item has no variable state. Hence, a call on one of its (nonnested) subprograms cannot normally have side effects. The only possible side effects from such a call would be through machine code insertions, imported subprograms,unchecked conversion to an access type declared within the subprogram, and similar features. The compiler may omit a call to such a subprogram even if such side effects exist, so the writer of such a subprogram has to keep this in mind.

Syntax

19 The form of a pragma Elaborate, Elaborate_All, or Elaborate_Body is as follows:

20 **pragma** Elaborate(*library_unit*_name{, *library_unit*_name});

21 **pragma** Elaborate_All(*library_unit*_name{, *library_unit*_name});

22 **pragma** Elaborate_Body[(*library_unit*_name)];

23 A pragma Elaborate or Elaborate_All is only allowed within a context_clause.

23.a **Ramification:** “Within a context_clause” allows it to be the last item in the context_clause. It can't be first, because the name has to denote something mentioned earlier.

24 {*library unit pragma (Elaborate_Body)* [partial]} {*pragma, library unit (Elaborate_Body)* [partial]} A pragma Elaborate_Body is a library unit pragma.

24.a **Discussion:** Hence, a pragma Elaborate or Elaborate_All is not elaborated, not that it makes any practical difference.

24.b Note that a pragma Elaborate or Elaborate_All is neither a program unit pragma, nor a library unit pragma.

Legality Rules

25 {*requires a completion (declaration to which a pragma Elaborate_Body applies)* [partial]} If a pragma Elaborate_Body applies to a declaration, then the declaration requires a completion [(a body)].

25.1/2 {*AI95-00217-06*} The *library_unit*_name of a pragma Elaborate or Elaborate_All shall denote a nonlimited view of a library unit.

25.a/2 **Reason:** These pragmas are intended to prevent elaboration check failures. But a limited view does not make anything visible that has an elaboration check, so the pragmas cannot do anything useful. Moreover, the pragmas would probably reintroduce the circularity that the limited_with_clause was intended to break. So we make such uses illegal.

Static Semantics

26 [A pragma Elaborate specifies that the body of the named library unit is elaborated before the current library_item. A pragma Elaborate_All specifies that each library_item that is needed by the named library unit declaration is elaborated before the current library_item. A pragma Elaborate_Body specifies that the body of the library unit is elaborated immediately after its declaration.]

26.a **Proof:** The official statement of the semantics of these pragmas is given in 10.2.

26.b **Implementation Note:** The presence of a pragma Elaborate_Body simplifies the removal of unnecessary Elaboration_Checks. For a subprogram declared immediately within a library unit to which a pragma Elaborate_Body applies, the only calls that can fail the Elaboration_Check are those that occur in the library unit itself, between the declaration and body of the called subprogram; if there are no such calls (which can easily be detected at compile time if there are no stubs), then no Elaboration_Checks are needed for that subprogram. The same is true for Elaboration_Checks on task activations and instantiations, and for library subprograms and generic units.

26.c **Ramification:** The fact that the unit of elaboration is the library_item means that if a subprogram_body is not a completion, it is impossible for any library_item to be elaborated between the declaration and the body of such a subprogram. Therefore, it is impossible for a call to such a subprogram to fail its Elaboration_Check.

26.d **Discussion:** The visibility rules imply that each *library_unit_*name of a pragma Elaborate or Elaborate_All has to denote a library unit mentioned by a previous with_clause of the same context_clause.

NOTES

27 12 A preelaborated library unit is allowed to have non-preelaborable children.

27.a/1 **Ramification:** {*8652/0035*} {*AI95-00002-01*} But generally not non-preelaborated subunits. (Non-preelaborated subunits of subprograms are allowed as discussed above.)

28 13 A library unit that is declared pure is allowed to have impure children.

28.a/1 **Ramification:** {*8652/0035*} {*AI95-00002-01*} But generally not impure subunits. (Impure subunits of subprograms are allowed as discussed above.)

28.b **Ramification:** Pragma Elaborate is mainly for closely related library units, such as when two package bodies 'with' each other's declarations. In such cases, Elaborate_All sometimes won't work.

Extensions to Ada 83

28.c {*extensions to Ada 83*} The concepts of preelaborability and purity are new to Ada 95. The Elaborate_All, Elaborate_Body, Preelaborate, and Pure pragmas are new to Ada 95.

28.d Pragmas Elaborate are allowed to be mixed in with the other things in the context_clause — in Ada 83, they were required to appear last.

Incompatibilities With Ada 95

28.e/2 {*AI95-00366-01*} {*incompatibilities with Ada 95*} The requirement that a partial view with available stream attributes be externally streamable can cause an incompatibility in rare cases. If there is a limited tagged type declared in a pure package with available attributes, and that type is used to declare a private extension in another pure package, and the full type for the private extension has a component of an explicitly limited record type, a protected type, or a type with access discriminants, then the stream attributes will have to be user-specified in the visible part of the package. That is not a requirement for Ada 95, but this combination seems very unlikely in pure packages. Note that this cannot be an incompatibility for a nonlimited type, as all of the types that are allowed in Ada 95 that would require explicitly defined stream attributes are limited (and thus cannot be used as components in a nonlimited type).

28.f/2 {*AI95-00403-01*} **Amendment Correction:** Added wording to cover missing cases for preelaborated generic units. This is incompatible as a preelaborated unit could have used a formal object to initialize a library-level object; that isn't allowed in Ada 2005. But such a unit wouldn't really be preelaborable, and Ada 95 compilers can reject such units (as this is a Binding Interpretation), so such units should be very rare.

Extensions to Ada 95

28.g/2 {*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** The concept of preelaborable initialization and pragma Preelaborable_Initialization are new. These allow more types of objects to be created in preelaborable units, and fix holes in the old rules.

28.h/2 {*AI95-00366-01*} Access-to-subprogram types and access-to-object types with a Storage_Size of 0 are allowed in pure units. The permission to omit calls was adjusted accordingly (which also fixes a hole in Ada 95, as access parameters are allowed, and changes in the values accessed by them must be taken into account).

Wording Changes from Ada 95

28.i/2 {*AI95-00002-01*} **Corrigendum:** The wording was changed so that subunits of a preelaborated subprogram are also preelaborated.

28.j/2 {*AI95-00217-06*} Disallowed pragma Elaborate and Elaborate_All for packages that are mentioned in a limited_with_clause.

Section 11: Exceptions

1 [This section defines the facilities for dealing with errors or other exceptional situations that arise during program execution.] {*exception occurrence*} {*condition: See also exception*} {*signal (an exception): See raise*} {*throw (an exception): See raise*} {*catch (an exception): See handle*} {*Exception*} [Glossary Entry]An *exception* represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an *exception occurrence*. [{*raise (an exception)* [partial]} To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. {*handle (an exception)* [partial]} Performing some actions in response to the arising of an exception is called *handling* the exception.]

1.a **To be honest:** {*handle (an exception occurrence)* [partial]} ...or handling the exception occurrence.

1.b **Ramification:** For example, an exception End_Error might represent error situations in which an attempt is made to read beyond end-of-file. During the execution of a partition, there might be numerous occurrences of this exception.

1.c **To be honest:** {*occurrence (of an exception)*} When the meaning is clear from the context, we sometimes use "*occurrence*" as a short-hand for "exception occurrence."

2 [An exception_declaration declares a name for an exception. An exception is raised initially either by a raise_statement or by the failure of a language-defined check. When an exception arises, control can be transferred to a user-provided exception_handler at the end of a handled_sequence_of_statements, or it can be propagated to a dynamically enclosing execution.]

Wording Changes from Ada 83

2.a We are more explicit about the difference between an exception and an occurrence of an exception. This is necessary because we now have a type (Exception_Occurrence) that represents exception occurrences, so the program can manipulate them. Furthermore, we say that when an exception is propagated, it is the same occurrence that is being propagated (as opposed to a new occurrence of the same exception). The same issue applies to a re-raise statement. In order to understand these semantics, we have to make this distinction.

11.1 Exception Declarations

1 {*exception*} An exception_declaration declares a name for an exception.

Syntax

2 exception_declaration ::= defining_identifier_list : **exception**;

Static Semantics

3 Each single exception_declaration declares a name for a different exception. If a generic unit includes an exception_declaration, the exception_declarations implicitly generated by different instantiations of the generic unit refer to distinct exceptions (but all have the same defining_identifier). The particular exception denoted by an exception name is determined at compilation time and is the same regardless of how many times the exception_declaration is elaborated.

3.a **Reason:** We considered removing this requirement inside generic bodies, because it is an implementation burden for implementations that wish to share code among several instances. In the end, it was decided that it would introduce too much implementation dependence.

3.b **Ramification:** Hence, if an exception_declaration occurs in a recursive subprogram, the exception name denotes the same exception for all invocations of the recursive subprogram. The reason for this rule is that we allow an exception occurrence to propagate out of its declaration's innermost containing master; if exceptions were created by their declarations like other entities, they would presumably be destroyed upon leaving the master; we would have to do something special to prevent them from propagating to places where they no longer exist.

3.c **Ramification:** Exception identities are unique across all partitions of a program.

4 {*predefined exception*} {*Constraint_Error (raised by failure of run-time check)*} {*Program_Error (raised by failure of run-time check)*} {*Storage_Error (raised by failure of run-time check)*} {*Tasking_Error (raised by failure of run-time check)*} The *predefined* exceptions are the ones declared in the declaration of package Standard: Constraint_Error, Program_Error, Storage_Error, and Tasking_Error[; one of them is raised when a language-defined check fails.]

4.a **Ramification:** The exceptions declared in the language-defined package IO_Exceptions, for example, are not predefined.

Dynamic Semantics

5 {*elaboration (exception_declaration)* [partial]} The elaboration of an exception_declaration has no effect.

6 {*Storage_Check* [partial]} {*check, language-defined (Storage_Check)*} {*Storage_Error (raised by failure of run-time check)*} The execution of any construct raises Storage_Error if there is insufficient storage for that execution. {*unspecified* [partial]} The amount of storage needed for the execution of constructs is unspecified.

6.a **Ramification:** Note that any execution whatsoever can raise Storage_Error. This allows much implementation freedom in storage management.

Examples

7 *Examples of user-defined exception declarations:*

8
```
Singular : exception;
Error    : exception;
Overflow, Underflow : exception;
```

Inconsistencies With Ada 83

8.a {*inconsistencies with Ada 83*} The exception Numeric_Error is now defined in the Obsolescent features Annex, as a rename of Constraint_Error. All checks that raise Numeric_Error in Ada 83 instead raise Constraint_Error in Ada 95. To increase upward compatibility, we also changed the rules to allow the same exception to be named more than once by a given handler. Thus, "**when** Constraint_Error | Numeric_Error =>" will remain legal in Ada 95, even though Constraint_Error and Numeric_Error now denote the same exception. However, it will not be legal to have separate handlers for Constraint_Error and Numeric_Error. This change is inconsistent in the rare case that an existing program explicitly raises Numeric_Error at a point where there is a handler for Constraint_Error; the exception will now be caught by that handler.

Wording Changes from Ada 83

8.b We explicitly define elaboration for exception_declarations.

11.2 Exception Handlers

1 [The response to one or more exceptions is specified by an exception_handler.]

Syntax

2
```
handled_sequence_of_statements ::=
     sequence_of_statements
  [exception
     exception_handler
    {exception_handler}]
```

3
```
exception_handler ::=
  when [choice_parameter_specification:] exception_choice {| exception_choice} =>
     sequence_of_statements
```

4 choice_parameter_specification ::= defining_identifier

5 exception_choice ::= *exception*_name | **others**

To be honest: {*handler*} "*Handler*" is an abbreviation for "exception_handler." 5.a

{*choice (of an exception_handler)*} Within this section, we sometimes abbreviate "exception_choice" to "*choice*." 5.b

Legality Rules

{*cover (of a choice and an exception)*} A choice with an *exception_*name *covers* the named exception. A choice with **others** covers all exceptions not named by previous choices of the same handled_sequence_of_statements. Two choices in different exception_handlers of the same handled_sequence_of_statements shall not cover the same exception. 6

Ramification: Two exception_choices of the same exception_handler may cover the same exception. For example, given two renaming declarations in separate packages for the same exception, one may nevertheless write, for example, "**when** Ada.Text_IO.Data_Error | My_Seq_IO.Data_Error =>". 6.a

An **others** choice even covers exceptions that are not visible at the place of the handler. Since exception raising is a dynamic activity, it is entirely possible for an **others** handler to handle an exception that it could not have named. 6.b

A choice with **others** is allowed only for the last handler of a handled_sequence_of_statements and as the only choice of that handler. 7

An *exception_*name of a choice shall not denote an exception declared in a generic formal package. 8

Reason: This is because the compiler doesn't know the identity of such an exception, and thus can't enforce the coverage rules. 8.a

Static Semantics

{*choice parameter*} A choice_parameter_specification declares a *choice parameter*, which is a constant object of type Exception_Occurrence (see 11.4.1). During the handling of an exception occurrence, the choice parameter, if any, of the handler represents the exception occurrence that is being handled. 9

Dynamic Semantics

{*execution (handled_sequence_of_statements)* [partial]} The execution of a handled_sequence_of_statements consists of the execution of the sequence_of_statements. [The optional handlers are used to handle any exceptions that are propagated by the sequence_of_statements.] 10

Examples

Example of an exception handler: 11

12
```
begin
   Open(File, In_File, "input.txt");   -- see A.8.2
exception
   when E : Name_Error =>
      Put("Cannot open input file : ");
      Put_Line(Exception_Message(E));  -- see 11.4.1
      raise;
end;
```

Extensions to Ada 83

{*extensions to Ada 83*} The syntax rule for exception_handler is modified to allow a choice_parameter_specification. 12.a

{*AI95-00114-01*} Different exception_choices of the same exception_handler may cover the same exception. This allows for "when Numeric_Error | Constraint_Error =>" even though Numeric_Error is a rename of Constraint_Error. This also allows one to "with" two different I/O packages, and then write, for example, "when Ada.Text_IO.Data_Error | My_Seq_IO.Data_Error =>" even though these might both be renames of the same exception. 12.b/2

Wording Changes from Ada 83

12.c The syntax rule for handled_sequence_of_statements is new. These are now used in all the places where handlers are allowed. This obviates the need to explain (in Sections 5, 6, 7, and 9) what portions of the program are handled by the handlers. Note that there are more such cases in Ada 95.

12.d The syntax rule for choice_parameter_specification is new.

11.3 Raise Statements

1 [A raise_statement raises an exception.]

Syntax

2/2 {*AI95-00361-01*} raise_statement ::= **raise**;
      | **raise** *exception_*name [**with** *string_*expression];

Legality Rules

3 The name, if any, in a raise_statement shall denote an exception. {*re-raise statement*} A raise_statement with no *exception_*name (that is, a *re-raise statement*) shall be within a handler, but not within a body enclosed by that handler.

Name Resolution Rules

3.1/2 {*AI95-00361-01*} The expression, if any, in a raise_statement, is expected to be of type String.

Dynamic Semantics

4/2 {*AI95-00361-01*} {*raise (an exception)*} To *raise an exception* is to raise a new occurrence of that exception[, as explained in 11.4]. {*execution (raise_statement with an exception_name)* [partial]} For the execution of a raise_statement with an *exception_*name, the named exception is raised. [If a *string_*expression is present, the expression is evaluated and its value is associated with the exception occurrence.] {*execution (re-raise statement)* [partial]} For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised [again].

4.a.1/2 **Proof:** {*AI95-00361-01*} The definition of Exceptions.Exception_Message includes a statement that the string is returned (see 11.4.1). We describe the use of the string here so that we don't have an unexplained parameter in this subclause.

4.a **Implementation Note:** For a re-raise statement, the implementation does not create a new Exception_Occurrence, but instead propagates the same Exception_Occurrence value. This allows the original cause of the exception to be determined.

Examples

5 *Examples of raise statements:*

6/2
```
{AI95-00433-01} raise Ada.IO_Exceptions.Name_Error;   -- see A.13
raise Queue_Error with "Buffer Full"; -- see 9.11
```

7
```
raise;                                      -- re-raise the current exception
```

Wording Changes from Ada 83

7.a The fact that the name in a raise_statement has to denote an exception is not clear from RM83. Clearly that was the intent, since the italicized part of the syntax rules so indicate, but there was no explicit rule. RM83-1.5(11) doesn't seem to give the italicized parts of the syntax any force.

Extensions to Ada 95

7.b/2 {*AI95-00361-01*} {*extensions to Ada 95*} The syntax of a raise_statement is extended to include a string message. This is more convenient than calling Exceptions.Exception_Message (*exception_*name'Identity, *string_*expression), and should encourage the use of message strings when raising exceptions.

11.4 Exception Handling

1 [When an exception occurrence is raised, normal program execution is abandoned and control is transferred to an applicable exception_handler, if any. {*handle (an exception occurrence)*} To *handle* an exception occurrence is to respond to the exceptional event. {*propagate*} To *propagate* an exception occurrence is to raise it again in another context; that is, to fail to respond to the exceptional event in the present context.]

1.a **Ramification:** In other words, if the execution of a given construct raises an exception, but does not handle it, the exception is propagated to an enclosing execution (except in the case of a task_body).

1.b/1 Propagation involves re-raising the same exception occurrence. For example, calling an entry of an uncallable task raises Tasking_Error; this is not propagation.

Dynamic Semantics

2 {*dynamically enclosing (of one execution by another)*} {*execution (dynamically enclosing)*} Within a given task, if the execution of construct *a* is defined by this International Standard to consist (in part) of the execution of construct *b*, then while *b* is executing, the execution of *a* is said to *dynamically enclose* the execution of *b*. {*innermost dynamically enclosing*} The *innermost dynamically enclosing* execution of a given execution is the dynamically enclosing execution that started most recently.

2.a **To be honest:** {*included (one execution by another)*} {*execution (included by another execution)*} If the execution of *a* dynamically encloses that of *b*, then we also say that the execution of *b* is *included in* the execution of *a*.

2.b **Ramification:** Examples: The execution of an if_statement dynamically encloses the evaluation of the condition after the **if** (during that evaluation). (Recall that “execution” includes both “elaboration” and “evaluation”, as well as other executions.) The evaluation of a function call dynamically encloses the execution of the sequence_of_statements of the function body (during that execution). Note that, due to recursion, several simultaneous executions of the same construct can be occurring at once during the execution of a particular task.

2.c Dynamically enclosing is not defined across task boundaries; a task's execution does not include the execution of any other tasks.

2.d Dynamically enclosing is only defined for executions that are occurring at a given moment in time; if an if_statement is currently executing the sequence_of_statements after **then**, then the evaluation of the condition is no longer dynamically enclosed by the execution of the if_statement (or anything else).

3 {*raise (an exception occurrence)*} When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is *abandoned*; that is, any portions of the execution that have not yet taken place are not performed. The construct is first completed, and then left, as explained in 7.6.1. Then:

- 4 If the construct is a task_body, the exception does not propagate further;

4.a **Ramification:** When an exception is raised by the execution of a task_body, there is no dynamically enclosing execution, so the exception does not propagate any further. If the exception occurred during the activation of the task, then the activator raises Tasking_Error, as explained in 9.2, “Task Execution - Task Activation”, but we don't define that as propagation; it's a special rule. Otherwise (the exception occurred during the execution of the handled_sequence_of_statements of the task), the task silently disappears. Thus, abnormal termination of tasks is not always considered to be an error.

- 5 If the construct is the sequence_of_statements of a handled_sequence_of_statements that has a handler with a choice covering the exception, the occurrence is handled by that handler;
- 6 {*propagate (an exception occurrence by an execution, to a dynamically enclosing execution)*} Otherwise, the occurrence is *propagated* to the innermost dynamically enclosing execution, which means that the occurrence is raised again in that context.

6.a **To be honest:** {*propagate (an exception by an execution)*} {*propagate (an exception by a construct)*} As shorthands, we refer to the *propagation of an exception*, and the *propagation by a construct*, if the execution of the construct propagates an exception occurrence.

7 {*handle (an exception occurrence)*} {*execution (handler)* [partial]} {*elaboration (choice_parameter_specification)* [partial]} When an occurrence is *handled* by a given handler, the choice_parameter_specification, if any, is first elaborated, which creates the choice parameter and initializes it to the occurrence. Then, the sequence_of_statements of the handler is executed; this execution replaces the abandoned portion of the execution of the sequence_of_statements.

7.a/2 **Ramification:** {*AI95-00318-02*} This "replacement" semantics implies that the handler can do pretty much anything the abandoned sequence could do; for example, in a function, the handler can execute a return statement that applies to the function.

7.b **Ramification:** The rules for exceptions raised in library units, main subprograms and partitions follow from the normal rules, plus the semantics of the environment task described in Section 10 (for example, the environment task of a partition elaborates library units and calls the main subprogram). If an exception is propagated by the main subprogram, it is propagated to the environment task, which then terminates abnormally, causing the partition to terminate abnormally. Although abnormal termination of tasks is not necessarily an error, abnormal termination of a partition due to an exception *is* an error.

NOTES

8 1 Note that exceptions raised in a declarative_part of a body are not handled by the handlers of the handled_sequence_of_statements of that body.

11.4.1 The Package Exceptions

Static Semantics

1 The following language-defined library package exists:

2/2
```
{AI95-00362-01} {AI95-00400-01} {AI95-00438-01} with Ada.Streams;
package Ada.Exceptions is
    pragma Preelaborate(Exceptions);
    type Exception_Id is private;
    pragma Preelaborable_Initialization(Exception_Id);
    Null_Id : constant Exception_Id;
    function Exception_Name(Id : Exception_Id) return String;
    function Wide_Exception_Name(Id : Exception_Id) return Wide_String;
    function Wide_Wide_Exception_Name(Id : Exception_Id)
        return Wide_Wide_String;
```

3/2
```
{AI95-00362-01}     type Exception_Occurrence is limited private;
    pragma Preelaborable_Initialization(Exception_Occurrence);
    type Exception_Occurrence_Access is access all Exception_Occurrence;
    Null_Occurrence : constant Exception_Occurrence;
```

4/2
```
{AI95-00329-01}     procedure Raise_Exception(E : in Exception_Id;
                              Message : in String := "");
        pragma No_Return(Raise_Exception);
    function Exception_Message(X : Exception_Occurrence) return String;
    procedure Reraise_Occurrence(X : in Exception_Occurrence);
```

5/2
```
{AI95-00400-01}     function Exception_Identity(X : Exception_Occurrence)
                                return Exception_Id;
    function Exception_Name(X : Exception_Occurrence) return String;
        -- Same as Exception_Name(Exception_Identity(X)).
    function Wide_Exception_Name(X : Exception_Occurrence)
        return Wide_String;
        -- Same as Wide_Exception_Name(Exception_Identity(X)).
    function Wide_Wide_Exception_Name(X : Exception_Occurrence)
        return Wide_Wide_String;
        -- Same as Wide_Wide_Exception_Name(Exception_Identity(X)).
    function Exception_Information(X : Exception_Occurrence) return String;
```

6/2
```
{AI95-00438-01}     procedure Save_Occurrence(Target : out Exception_Occurrence;
                              Source : in Exception_Occurrence);
    function Save_Occurrence(Source : Exception_Occurrence)
                             return Exception_Occurrence_Access;
```

```
{AI95-00438-01}     procedure Read_Exception_Occurrence                6.1/2
       (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
        Item   : out Exception_Occurrence);
    procedure Write_Exception_Occurrence
       (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
        Item   : in Exception_Occurrence);
```

```
{AI95-00438-01}     for Exception_Occurrence'Read use Read_Exception_Occurrence;   6.2/2
    for Exception_Occurrence'Write use Write_Exception_Occurrence;
```

```
{AI95-00438-01} private                                                6.3/2
   ... -- not specified by the language
end Ada.Exceptions;
```

Each distinct exception is represented by a distinct value of type Exception_Id. Null_Id does not represent any exception, and is the default initial value of type Exception_Id. Each occurrence of an exception is represented by a value of type Exception_Occurrence. Null_Occurrence does not represent any exception occurrence, and is the default initial value of type Exception_Occurrence. 7

For a prefix E that denotes an exception, the following attribute is defined: 8/1

E'Identity E'Identity returns the unique identity of the exception. The type of this attribute is Exception_Id. 9

Ramification: In a distributed program, the identity is unique across an entire program, not just across a single partition. Exception propagation works properly across RPC's. An exception can be propagated from one partition to another, and then back to the first, where its identity is known. 9.a

{*AI95-00361-01*} Raise_Exception raises a new occurrence of the identified exception. 10/2

{*AI95-00361-01*} {*AI95-00378-01*} Exception_Message returns the message associated with the given Exception_Occurrence. For an occurrence raised by a call to Raise_Exception, the message is the Message parameter passed to Raise_Exception. For the occurrence raised by a raise_statement with an *exception*_name and a *string*_expression, the message is the *string*_expression. For the occurrence raised by a raise_statement with an *exception*_name but without a *string*_expression, the message is a string giving implementation-defined information about the exception occurrence. In all cases, Exception_Message returns a string with lower bound 1. 10.1/2

Implementation defined: The information returned by Exception_Message. 10.a

Ramification: Given an exception E, the raise_statement: 10.b

```
raise E;                                                              10.c
```

is equivalent to this call to Raise_Exception: 10.d

```
Raise_Exception(E'Identity, Message => implementation-defined-string);   10.e
```

{*AI95-00361-01*} Similarly, the raise_statement: 10.e.1/2

```
raise E with "some information";                                      10.e.2/2
```

is equivalent to this call to Raise_Exception: 10.e.3/2

```
Raise_Exception(E'Identity, Message => "some information");           10.e.4/2
```

{*AI95-00361-01*} Reraise_Occurrence reraises the specified exception occurrence. 10.2/2

Ramification: The following handler: 10.f

```
when others =>                                                        10.g
    Cleanup;
    raise;
```

is equivalent to this one: 10.h

```
when X : others =>                                                    10.i
    Cleanup;
    Reraise_Occurrence(X);
```

Exception_Identity returns the identity of the exception of the occurrence. 11

{*AI95-00400-01*} The Wide_Wide_Exception_Name functions return the full expanded name of the exception, in upper case, starting with a root library unit. For an exception declared immediately within 12/2 package Standard, the defining_identifier is returned. The result is implementation defined if the exception is declared within an unnamed block_statement.

12.a **Ramification:** See the Implementation Permission below.

12.b **To be honest:** This name, as well as each prefix of it, does not denote a renaming_declaration.

12.c/2 **Implementation defined:** The result of Exceptions.Wide_Wide_Exception_Name for exceptions declared within an unnamed block_statement.

12.d **Ramification:** Note that we're talking about the name of the exception, not the name of the occurrence.

12.1/2 {*AI95-00400-01*} The Exception_Name functions (respectively, Wide_Exception_Name) return the same sequence of graphic characters as that defined for Wide_Wide_Exception_Name, if all the graphic characters are defined in Character (respectively, Wide_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide_Wide_Exception_Name for the same value of the argument.

12.e/2 **Implementation defined:** The sequence of characters of the value returned by Exceptions.Exception_Name (respectively, Exceptions.Wide_Exception_Name) when some of the graphic characters of Exceptions.Wide_Wide_Exception_Name are not defined in Character (respectively, Wide_Character).

12.2/2 {*AI95-00378-01*} {*AI95-00417-01*} The string returned by the Exception_Name, Wide_Exception_Name, and Wide_Wide_Exception_Name functions has lower bound 1.

13/2 {*AI95-00378-01*} Exception_Information returns implementation-defined information about the exception occurrence. The returned string has lower bound 1.

13.a **Implementation defined:** The information returned by Exception_Information.

14/2 {*AI95-00241-01*} {*AI95-00446-01*} Reraise_Occurrence has no effect in the case of Null_Occurrence. {*Constraint_Error (raised by failure of run-time check)*} Raise_Exception and Exception_Name raise Constraint_Error for a Null_Id. Exception_Message, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Occurrence. Exception_Identity applied to Null_Occurrence returns Null_Id.

14.a.1/2 **Ramification:** {*AI95-00241-01*} Null_Occurrence can be tested for by comparing Exception_Identity(Occurrence) to Null_Id.

14.a.2/2 **Discussion:** {*AI95-00446-01*} Raise_Exception was changed so that it always raises an exception and thus can be a No_Return procedure. A similar change was not made for Reraise_Occurrence, as doing so was determined to be a significant incompatibility. It is not unusual to pass an Exception_Occurrence to other code to delay raising it. If there was no exception, passing Null_Occurrence works fine (nothing is raised). Moreover, as there is no test for Null_Occurrence in Ada 95, this is the only way to write such code without using additional flags. Breaking this sort of code is unacceptable.

15 The Save_Occurrence procedure copies the Source to the Target. The Save_Occurrence function uses an allocator of type Exception_Occurrence_Access to create a new object, copies the Source to this new object, and returns an access value designating this new object; [the result may be deallocated using an instance of Unchecked_Deallocation.]

15.a **Ramification:** It's OK to pass Null_Occurrence to the Save_Occurrence subprograms; they don't raise an exception, but simply save the Null_Occurrence.

15.1/2 {*AI95-00438-01*} Write_Exception_Occurrence writes a representation of an exception occurrence to a stream; Read_Exception_Occurrence reconstructs an exception occurrence from a stream (including one written in a different partition).

15.b/2 **Ramification:** This routines are used to define the stream attributes (see 13.13.2) for Exception_Occurrence.

15.c/2 The identity of the exception, as well as the Exception_Name and Exception_Message, have to be preserved across partitions.

15.d/2 The string returned by Exception_Name or Exception_Message on the result of calling the Read attribute on a given stream has to be the same as the value returned by calling the corresponding function on the exception occurrence that was written into the stream with the Write attribute. The string returned by Exception_Information need not be the same, since it is implementation defined anyway.

Reason: This is important for supporting writing exception occurrences to external files for post-mortem analysis, as well as propagating exceptions across remote subprogram calls in a distributed system (see E.4). 15.e/2

Implementation Requirements

This paragraph was deleted.{*AI95-00438-01*} 16/2

This paragraph was deleted. 16.a/2

This paragraph was deleted. 16.b/2

This paragraph was deleted. 16.c/2

Implementation Permissions

An implementation of Exception_Name in a space-constrained environment may return the defining_-identifier instead of the full expanded name. 17

The string returned by Exception_Message may be truncated (to no less than 200 characters) by the Save_Occurrence procedure [(not the function)], the Reraise_Occurrence procedure, and the re-raise statement. 18

Reason: The reason for allowing truncation is to ease implementations. The reason for choosing the number 200 is that this is the minimum source line length that implementations have to support, and this feature seems vaguely related since it's usually a “one-liner”. Note that an implementation is allowed to do this truncation even if it supports arbitrarily long lines. 18.a

Implementation Advice

Exception_Message (by default) and Exception_Information should produce information useful for debugging. Exception_Message should be short (about one line), whereas Exception_Information can be long. Exception_Message should not include the Exception_Name. Exception_Information should include both the Exception_Name and the Exception_Message. 19

Implementation Advice: Exception_Information should provide information useful for debugging, and should include the Exception_Name and Exception_Message. 19.a.1/2

Implementation Advice: Exception_Message by default should be short, provide information useful for debugging, and should not include the Exception_Name. 19.a.2/2

Reason: It may seem strange to define two subprograms whose semantics is implementation defined. The idea is that a program can print out debugging/error-logging information in a portable way. The program is portable in the sense that it will work in any implementation; it might print out different information, but the presumption is that the information printed out is appropriate for debugging/error analysis on that system. 19.a

Implementation Note: As an example, Exception_Information might include information identifying the location where the exception occurred, and, for predefined exceptions, the specific kind of language-defined check that failed. There is an implementation trade-off here, between how much information is represented in an Exception_Occurrence, and how much can be passed through a re-raise. 19.b

The string returned should be in a form suitable for printing to an error log file. This means that it might need to contain line-termination control characters with implementation-defined I/O semantics. The string should neither start nor end with a newline. 19.c

If an implementation chooses to provide additional functionality related to exceptions and their occurrences, it should do so by providing one or more children of Ada.Exceptions. 19.d

Note that exceptions behave as if declared at library level; there is no “natural scope” for an exception; an exception always exists. Hence, there is no harm in saving an exception occurrence in a data structure, and reraising it later. The reraise has to occur as part of the same program execution, so saving an exception occurrence in a file, reading it back in from a different program execution, and then reraising it is not required to work. This is similar to I/O of access types. Note that it is possible to use RPC to propagate exceptions across partitions. 19.e

Here's one way to implement Exception_Occurrence in the private part of the package. Using this method, an implementation need store only the actual number of characters in exception messages. If the user always uses small messages, then exception occurrences can be small. If the user never uses messages, then exception occurrences can be smaller still: 19.f

19.g
```
    type Exception_Occurrence(Message_Length : Natural := 200) is
        limited record
            Id : Exception_Id;
            Message : String(1..Message_Length);
        end record;
```

19.h At the point where an exception is raised, an Exception_Occurrence can be allocated on the stack with exactly the right amount of space for the message — none for an empty message. This is just like declaring a constrained object of the type:

19.i
```
    Temp : Exception_Occurrence(10); -- for a 10-character message
```

19.j After finding the appropriate handler, the stack can be cut back, and the Temp copied to the right place. This is similar to returning an unknown-sized object from a function. It is not necessary to allocate the maximum possible size for every Exception_Occurrence. If, however, the user declares an Exception_Occurrence object, the discriminant will be permanently set to 200. The Save_Occurrence procedure would then truncate the Exception_Message. Thus, nothing is lost until the user tries to save the occurrence. If the user is willing to pay the cost of heap allocation, the Save_Occurrence function can be used instead.

19.k Note that any arbitrary-sized implementation-defined Exception_Information can be handled in a similar way. For example, if the Exception_Occurrence includes a stack traceback, a discriminant can control the number of stack frames stored. The traceback would be truncated or entirely deleted by the Save_Occurrence procedure — as the implementation sees fit.

19.l If the internal representation involves pointers to data structures that might disappear, it would behoove the implementation to implement it as a controlled type, so that assignment can either copy the data structures or else null out the pointers. Alternatively, if the data structures being pointed at are in a task control block, the implementation could keep a unique sequence number for each task, so it could tell when a task's data structures no longer exist.

19.m Using the above method, heap space is never allocated unless the user calls the Save_Occurrence function.

19.n An alternative implementation would be to store the message strings on the heap when the exception is raised. (It could be the global heap, or it could be a special heap just for this purpose — it doesn't matter.) This representation would be used only for choice parameters. For normal user-defined exception occurrences, the Save_Occurrence procedure would copy the message string into the occurrence itself, truncating as necessary. Thus, in this implementation, Exception_Occurrence would be implemented as a variant record:

19.o
```
    type Exception_Occurrence_Kind is (Normal, As_Choice_Param);
```

19.p
```
    type Exception_Occurrence(Kind : Exception_Occurrence_Kind := Normal) is
        limited record
            case Kind is
                when Normal =>
                    ... -- space for 200 characters
                when As_Choice_Param =>
                    ... -- pointer to heap string
            end case;
        end record;
```

19.q Exception_Occurrences created by the run-time system during exception raising would be As_Choice_Param. User-declared ones would be Normal — the user cannot see the discriminant, and so cannot set it to As_Choice_Param. The strings in the heap would be freed upon completion of the handler.

19.r This alternative implementation corresponds to a heap-based implementation of functions returning unknown-sized results.

19.s One possible implementation of Reraise_Occurrence is as follows:

19.t
```
    procedure Reraise_Occurrence(X : in Exception_Occurrence) is
    begin
        Raise_Exception(Identity(X), Exception_Message(X));
    end Reraise_Occurrence;
```

19.u However, some implementations may wish to retain more information across a re-raise — a stack traceback, for example.

19.v **Ramification:** Note that Exception_Occurrence is a definite subtype. Hence, values of type Exception_Occurrence may be written to an error log for later analysis, or may be passed to subprograms for immediate error analysis.

19.w/2 *This paragraph was deleted.*{*AI95-00400-01*}

Extensions to Ada 83

19.x {*extensions to Ada 83*} The Identity attribute of exceptions is new, as is the package Exceptions.

Inconsistencies With Ada 95

{*AI95-00241-01*} {*inconsistencies with Ada 95*} **Amendment Correction:** Exception_Identity of an Exception_Occurrence now is defined to return Null_Id for Null_Occurrence, rather than raising Constraint_Error. This provides a simple way to test for Null_Occurrence. We expect that programs that need Constraint_Error raised will be very rare; they can be easily fixed by explicitly testing for Null_Id or by using Exception_Name instead. 19.y/2

{*AI95-00378-01*} {*AI95-00417-01*} **Amendment Correction:** We now define the lower bound of the string returned from [[Wide_]Wide_]Exception_Name, Exception_Message, and Exception_Information. This makes working with the returned string easier, and is consistent with many other string-returning functions in Ada. This is technically an inconsistency; if a program depended on some other lower bound for the string returned from one of these functions, it could fail when compiled with Ada 2005. Such code is not portable even between Ada 95 implementations, so it should be very rare. 19.z/2

{*AI95-00446-01*} **Amendment Correction:** Raise_Exception now raises Constraint_Error if passed Null_Id. This means that it always raises an exception, and thus we can apply pragma No_Return to it. We expect that programs that call Raise_Exception with Null_Id will be rare, and programs that do that and expect no exception to be raised will be rarer; such programs can be easily fixed by explicitly testing for Null_Id before calling Raise_Exception. 19.aa/2

Incompatibilities With Ada 95

{*AI95-00400-01*} {*AI95-00438-01*} {*incompatibilities with Ada 95*} Functions Wide_Exception_Name and Wide_Wide_Exception_Name, and procedures Read_Exception_Occurrence and Write_Exception_Occurrence are newly added to Exceptions. If Exceptions is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Exceptions is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 19.bb/2

Extensions to Ada 95

{*AI95-00362-01*} {*extensions to Ada 95*} The package Exceptions is preelaborated, and types Exception_Id and Exception_Occurrence have preelaborable initialization, allowing this package to be used in preelaborated units. 19.cc/2

Wording Changes from Ada 95

{*AI95-00361-01*} The meaning of Exception_Message is reworded to reflect that the string can come from a raise_statement as well as a call of Raise_Exception. 19.dd/2

{*AI95-00400-01*} Added Wide_Exception_Name and Wide_Wide_Exception_Name because identifiers can now contain characters outside of Latin-1. 19.ee/2

11.4.2 Pragmas Assert and Assertion_Policy

{*AI95-00286-01*} Pragma Assert is used to assert the truth of a Boolean expression at any point within a sequence of declarations or statements. Pragma Assertion_Policy is used to control whether such assertions{*Assertions*} are to be ignored by the implementation, checked at run-time, or handled in some implementation-defined manner. 1/2

Syntax

{*AI95-00286-01*} The form of a pragma Assert is as follows: 2/2

pragma Assert([Check =>] *boolean*_expression[, [Message =>] *string*_expression]); 3/2

A pragma Assert is allowed at the place where a declarative_item or a statement is allowed. 4/2

{*AI95-00286-01*} The form of a pragma Assertion_Policy is as follows: 5/2

pragma Assertion_Policy(*policy*_identifier); 6/2

{*configuration pragma (Assertion_Policy)* [partial]} {*pragma, configuration (Assertion_Policy)* [partial]} A pragma Assertion_Policy is a configuration pragma. 7/2

Name Resolution Rules

{*AI95-00286-01*} The expected type for the *boolean*_expression of a pragma Assert is any boolean type. The expected type for the *string*_expression of a pragma Assert is type String. 8/2

8.a/2 **Reason:** We allow any boolean type to be like if_statements and other conditionals; we only allow String for the message in order to match raise_statements.

Legality Rules

9/2 {*AI95-00286-01*} The *policy*_identifier of a pragma Assertion_Policy shall be either Check, Ignore, or an implementation-defined identifier.

9.a/2 **Implementation defined:** Implementation-defined *policy*_identifiers allowed in a pragma Assertion_Policy.

Static Semantics

10/2 {*AI95-00286-01*} A pragma Assertion_Policy is a configuration pragma that specifies the assertion policy in effect for the compilation units to which it applies. Different policies may apply to different compilation units within the same partition. The default assertion policy is implementation-defined.

10.a/2 **Implementation defined:** The default assertion policy.

11/2 {*AI95-00286-01*} The following language-defined library package exists:

```
12/2   package Ada.Assertions is
          pragma Pure(Assertions);
13/2      Assertion_Error : exception;
14/2      procedure Assert(Check : in Boolean);
          procedure Assert(Check : in Boolean; Message : in String);
15/2   end Ada.Assertions;
```

16/2 {*AI95-00286-01*} A compilation unit containing a pragma Assert has a semantic dependence on the Assertions library unit.

17/2 {*AI95-00286-01*} The assertion policy that applies to a generic unit also applies to all its instances.

Dynamic Semantics

18/2 {*AI95-00286-01*} An assertion policy {*assertion policy*} specifies how a pragma Assert is interpreted by the implementation. If the assertion policy is Ignore at the point of a pragma Assert, the pragma is ignored. If the assertion policy is Check at the point of a pragma Assert, the elaboration of the pragma consists of evaluating the boolean expression, and if the result is False, evaluating the Message argument, if any, and raising the exception Assertions.Assertion_Error, with a message if the Message argument is provided.

19/2 {*AI95-00286-01*} Calling the procedure Assertions.Assert without a Message parameter is equivalent to:

```
20/2   if Check = False then
          raise Ada.Assertions.Assertion_Error;
       end if;
```

21/2 {*AI95-00286-01*} Calling the procedure Assertions.Assert with a Message parameter is equivalent to:

```
22/2   if Check = False then
          raise Ada.Assertions.Assertion_Error with Message;
       end if;
```

23/2 {*AI95-00286-01*} The procedures Assertions.Assert have these effects independently of the assertion policy in effect.

Implementation Permissions

24/2 {*AI95-00286-01*} Assertion_Error may be declared by renaming an implementation-defined exception from another package.

24.a/2 **Reason:** This permission is intended to allow implementations which had an implementation-defined Assert pragma to continue to use their originally defined exception. Without this permission, such an implementation would be incorrect, as Exception_Name would return the wrong name.

{*AI95-00286-01*} Implementations may define their own assertion policies. 25/2

NOTES

2 {*AI95-00286-01*} Normally, the boolean expression in a pragma Assert should not call functions that have significant side-effects when the result of the expression is True, so that the particular assertion policy in effect will not affect normal operation of the program. 26/2

Extensions to Ada 95

{*AI95-00286-01*} {*extensions to Ada 95*} Pragmas Assert and Assertion_Policy, and package Assertions are new. 26.a/2

11.4.3 Example of Exception Handling

Examples

Exception handling may be used to separate the detection of an error from the response to that error: 1

```
{AI95-00433-01} package File_System is                                     2/2
    type File_Handle is limited private;

    File_Not_Found : exception;                                            3
    procedure Open(F : in out File_Handle; Name : String);
        -- raises File_Not_Found if named file does not exist

    End_Of_File : exception;                                               4
    procedure Read(F : in out File_Handle; Data : out Data_Type);
        -- raises End_Of_File if the file is not open

    ...                                                                    5
end File_System;

{AI95-00433-01} package body File_System is                                6/2
    procedure Open(F : in out File_Handle; Name : String) is
    begin
        if File_Exists(Name) then
            ...
        else
            raise File_Not_Found with "File not found: " & Name & ".";
        end if;
    end Open;

    procedure Read(F : in out File_Handle; Data : out Data_Type) is       7
    begin
        if F.Current_Position <= F.Last_Position then
            ...
        else
            raise End_Of_File;
        end if;
    end Read;

    ...                                                                    8

end File_System;                                                           9
```

10
```
with Ada.Text_IO;
with Ada.Exceptions;
with File_System; use File_System;
use Ada;
procedure Main is
begin
    ... -- call operations in File_System
exception
    when End_Of_File =>
        Close(Some_File);
    when Not_Found_Error : File_Not_Found =>
        Text_IO.Put_Line(Exceptions.Exception_Message(Not_Found_Error));
    when The_Error : others =>
        Text_IO.Put_Line("Unknown error:");
        if Verbosity_Desired then
            Text_IO.Put_Line(Exceptions.Exception_Information(The_Error));
        else
            Text_IO.Put_Line(Exceptions.Exception_Name(The_Error));
            Text_IO.Put_Line(Exceptions.Exception_Message(The_Error));
        end if;
        raise;
end Main;
```

11 In the above example, the File_System package contains information about detecting certain exceptional situations, but it does not specify how to handle those situations. Procedure Main specifies how to handle them; other clients of File_System might have different handlers, even though the exceptional situations arise from the same basic causes.

Wording Changes from Ada 83

11.a The sections labeled “Exceptions Raised During ...” are subsumed by this clause, and by parts of Section 9.

11.5 Suppressing Checks

1/2 {*AI95-00224-01*} *Checking pragmas*{*Checking pragmas*} give instructions to an implementation on handling language-defined checks. A pragma Suppress gives permission to an implementation to omit certain language-defined checks, while a pragma Unsuppress revokes the permission to omit checks..

2 {*language-defined check*} {*check (language-defined)*} {*run-time check: See language-defined check*} {*run-time error*} {*error (run-time)*} A *language-defined check* (or simply, a “check”) is one of the situations defined by this International Standard that requires a check to be made at run time to determine whether some condition is true. {*failure (of a language-defined check)*} A check *fails* when the condition being checked is false, causing an exception to be raised.

2.a **Discussion:** All such checks are defined under “Dynamic Semantics” in clauses and subclauses throughout the standard.

Syntax

3/2 {*AI95-00224-01*} The forms of checking pragmas are as follows:

4/2 {*AI95-00224-01*} **pragma** Suppress(identifier);

4.1/2 {*AI95-00224-01*} **pragma** Unsuppress(identifier);

5/2 {*AI95-00224-01*} {*configuration pragma (Suppress)* [partial]} {*pragma, configuration (Suppress)* [partial]} {*configuration pragma (Unsuppress)* [partial]} {*pragma, configuration (Unsuppress)* [partial]} A checking pragma is allowed only immediately within a declarative_part, immediately within a package_-specification, or as a configuration pragma.

Legality Rules

6/2 {*AI95-00224-01*} The identifier shall be the name of a check.

This paragraph was deleted.{*AI95-00224-01*} 7/2

Static Semantics

{*AI95-00224-01*} A checking pragma applies to the named check in a specific region, and applies to all entities in that region. A checking pragma given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a checking pragma given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies. 7.1/2

{*AI95-00224-01*} If a checking pragma applies to a generic instantiation, then the checking pragma also applies to the instance. If a checking pragma applies to a call to a subprogram that has a pragma Inline applied to it, then the checking pragma also applies to the inlined subprogram body. 7.2/2

{*AI95-00224-01*} A pragma Suppress gives permission to an implementation to omit the named check (or every check in the case of All_Checks) for any entities to which it applies. {*suppressed check*} If permission has been given to suppress a given check, the check is said to be *suppressed*. 8/2

> **Ramification:** A check is suppressed even if the implementation chooses not to actually generate better code. {*Program_Error (raised by failure of run-time check)*} This allows the implementation to raise Program_Error, for example, if the erroneousness is detected. 8.a

{*AI95-00224-01*} A pragma Unsuppress revokes the permission to omit the named check (or every check in the case of All_Checks) given by any pragma Suppress that applies at the point of the pragma Unsuppress. The permission is revoked for the region to which the pragma Unsuppress applies. If there is no such permission at the point of a pragma Unsuppress, then the pragma has no effect. A later pragma Suppress can renew the permission. 8.1/2

The following are the language-defined checks: 9

- {*Constraint_Error (raised by failure of run-time check)*} [The following checks correspond to situations in which the exception Constraint_Error is raised upon failure.] 10

  {*8652/0036*} {*AI95-00176-01*} {*AI95-00231-01*} {*Access_Check* [distributed]} Access_Check 11/2
  [When evaluating a dereference (explicit or implicit), check that the value of the name is not **null**. When converting to a subtype that excludes null, check that the converted value is not **null**.]

  {*Discriminant_Check* [distributed]} Discriminant_Check [Check that the discriminants of a composite value have the values imposed by a discriminant constraint. Also, when accessing a record component, check that it exists for the current discriminant values.] 12

  {*AI95-00434-01*} {*Division_Check* [distributed]} Division_Check 13/2
  [Check that the second operand is not zero for the operations /, **rem** and **mod**.]

  {*Index_Check* [distributed]} Index_Check 14
  [Check that the bounds of an array value are equal to the corresponding bounds of an index constraint. Also, when accessing a component of an array object, check for each dimension that the given index value belongs to the range defined by the bounds of the array object. Also, when accessing a slice of an array object, check that the given discrete range is compatible with the range defined by the bounds of the array object.]

  {*Length_Check* [distributed]} Length_Check 15
  [Check that two arrays have matching components, in the case of array subtype conversions, and logical operators for arrays of boolean components.]

  {*Overflow_Check* [distributed]} Overflow_Check [Check that a scalar value is within the base range of its type, in cases where the implementation chooses to raise an exception instead of returning the correct mathematical result.] 16

17 {*Range_Check* [distributed]} Range_Check
[Check that a scalar value satisfies a range constraint. Also, for the elaboration of a subtype_indication, check that the constraint (if present) is compatible with the subtype denoted by the subtype_mark. Also, for an aggregate, check that an index or discriminant value belongs to the corresponding subtype. Also, check that when the result of an operation yields an array, the value of each component belongs to the component subtype.]

18 {*Tag_Check* [distributed]} Tag_Check
[Check that operand tags in a dispatching call are all equal. Check for the correct tag on tagged type conversions, for an assignment_statement, and when returning a tagged limited object from a function.]

19 • {*Program_Error (raised by failure of run-time check)*} [The following checks correspond to situations in which the exception Program_Error is raised upon failure.]

19.1/2 {*AI95-00280*} {*Accessibility_Check* [distributed]} Accessibility_Check
[Check the accessibility level of an entity or view.]

19.2/2 {*AI95-00280*} {*Allocation_Check* [distributed]} Allocation_Check
[For an allocator, check that the master of any tasks to be created by the allocator is not yet completed or some dependents have not yet terminated, and that the finalization of the collection has not started.]

20 {*Elaboration_Check* [distributed]} Elaboration_Check [When a subprogram or protected entry is called, a task activation is accomplished, or a generic instantiation is elaborated, check that the body of the corresponding unit has already been elaborated.]

21/2 *This paragraph was deleted.*{*AI95-00280*}

22 • [The following check corresponds to situations in which the exception Storage_Error is raised upon failure.]

23 {*Storage_Check* [distributed]} {*Storage_Error (raised by failure of run-time check)*} Storage_Check
[Check that evaluation of an allocator does not require more space than is available for a storage pool. Check that the space available for a task or subprogram has not been exceeded.]

23.a **Reason:** We considered splitting this out into three categories: Pool_Check (for allocators), Stack_Check (for stack usage), and Heap_Check (for implicit use of the heap — use of the heap other than through an allocator). Storage_Check would then represent the union of these three. However, there seems to be no compelling reason to do this, given that it is not feasible to split Storage_Error.

24 • [The following check corresponds to all situations in which any predefined exception is raised.]

25 {*All_Checks* [distributed]} All_Checks
Represents the union of all checks; [suppressing All_Checks suppresses all checks.]

25.a **Ramification:** All_Checks includes both language-defined and implementation-defined checks.

Erroneous Execution

26 {*erroneous execution (cause)* [partial]} If a given check has been suppressed, and the corresponding error situation occurs, the execution of the program is erroneous.

Implementation Permissions

27/2 {*AI95-00224-01*} An implementation is allowed to place restrictions on checking pragmas, subject only to the requirement that pragma Unsuppress shall allow any check names supported by pragma Suppress. An implementation is allowed to add additional check names, with implementation-defined semantics. {*unspecified* [partial]} When Overflow_Check has been suppressed, an implementation may also suppress an unspecified subset of the Range_Checks.

This paragraph was deleted.{*AI95-00224-01*} 27.a/2

Implementation defined: Implementation-defined check names. 27.b

Discussion: For Overflow_Check, the intention is that the implementation will suppress any Range_Checks that are implemented in the same manner as Overflow_Checks (unless they are free). 27.c

{*AI95-00224-01*} An implementation may support an additional parameter on pragma Unsuppress similar to the one allowed for pragma Suppress (see J.10). The meaning of such a parameter is implementation-defined. 27.1/2

Implementation defined: Existence and meaning of second parameter of pragma Unsuppress. 27.c.1/2

Implementation Advice

The implementation should minimize the code executed for checks that have been suppressed. 28

Implementation Advice: Code executed for checks that have been suppressed should be minimized. 28.a.1/2

Implementation Note: However, if a given check comes for free (for example, the hardware automatically performs the check in parallel with doing useful work) or nearly free (for example, the check is a tiny portion of an expensive run-time system call), the implementation should not bother to suppress the check. Similarly, if the implementation detects the failure at compile time and provides a warning message, there is no need to actually suppress the check. 28.a

NOTES

3 {*optimization*} {*efficiency*} There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress should be used only for efficiency reasons. 29

4 {*AI95-00224-01*} It is possible to give both a pragma Suppress and Unsuppress for the same check immediately within the same declarative_part. In that case, the last pragma given determines whether or not the check is suppressed. Similarly, it is possible to resuppress a check which has been unsuppressed by giving a pragma Suppress in an inner declarative region. 29.1/2

Examples

{*AI95-00224-01*} *Examples of suppressing and unsuppressing checks:* 30/2

31/2
```
{AI95-00224-01} pragma Suppress(Index_Check);
pragma Unsuppress(Overflow_Check);
```

Extensions to Ada 83

{*extensions to Ada 83*} A pragma Suppress is allowed as a configuration pragma. A pragma Suppress without a name is allowed in a package_specification. 31.a

Additional check names are added. We allow implementations to define their own checks. 31.b

Wording Changes from Ada 83

We define the checks in a distributed manner. Therefore, the long list of what checks apply to what is merely a NOTE. 31.c

We have removed the detailed rules about what is allowed in a pragma Suppress, and allow implementations to invent their own. The RM83 rules weren't quite right, and such a change is necessary anyway in the presence of implementation-defined checks. 31.d

We make it clear that the difference between a Range_Check and an Overflow_Check is fuzzy. This was true in Ada 83, given RM83-11.6, but it was not clear. We considered removing Overflow_Check from the language or making it obsolescent, just as we did for Numeric_Error. However, we kept it for upward compatibility, and because it may be useful on machines where range checking costs more than overflow checking, but overflow checking still costs something. Different compilers will suppress different checks when asked to suppress Overflow_Check — the non-uniformity in this case is not harmful, and removing it would have a serious impact on optimizers. 31.e

Under Access_Check, dereferences cover the cases of selected_component, indexed_component, slice, and attribute that are listed in RM83, as well as the new explicit_dereference, which was included in selected_component in RM83. 31.f

Extensions to Ada 95

{*AI95-00224-01*} {*extensions to Ada 95*} Pragma Unsuppress is new. 31.g/2

{*AI95-00280-01*} Allocation_Check was added to support suppressing the new check on allocators (see 4.8). 31.h/2

Wording Changes from Ada 95

31.i/2 {*8652/0036*} {*AI95-00176-01*} {*AI95-00224-01*} The description of Access_Check was corrected by the Corrigendum to include the discriminant case. This change was then replaced by the more general notion of checking conversions to subtypes that exclude null in Ada 2005.

31.j/2 {*AI95-00224-01*} The On parameter of pragma Suppress was moved to Annex J. This feature's effect is inherently non-portable, depending on the implementation's model of computation. Compiler surveys demonstrated this, showing that implementations vary widely in the interpretation of these parameters, even on the same target. While this is relatively harmless for Suppress (which is never required to do anything), it would be a significant problem for Unsuppress (we want the checks to be made for all implementations). By moving it, we avoid needing to define the meaning of Unsuppress with an On parameter.

31.k/2 {*AI95-00280-01*} The order of the Program_Error checks was corrected to be alphabetical.

11.6 Exceptions and Optimization

1 [{*language-defined check*} {*check (language-defined)*} {*run-time error*} {*error (run-time)*} {*optimization*} {*efficiency*} This clause gives permission to the implementation to perform certain “optimizations” that do not necessarily preserve the canonical semantics.]

Dynamic Semantics

2 {*canonical semantics*} The rest of this International Standard (outside this clause) defines the *canonical semantics* of the language. [The canonical semantics of a given (legal) program determines a set of possible external effects that can result from the execution of the program with given inputs.]

2.a **Ramification:** Note that the canonical semantics is a set of possible behaviors, since some reordering, parallelism, and non-determinism is allowed by the canonical semantics.

2.b **Discussion:** The following parts of the canonical semantics are of particular interest to the reader of this clause:

- 2.c Behavior in the presence of abnormal objects and objects with invalid representations (see 13.9.1).
- 2.d Various actions that are defined to occur in an arbitrary order.
- 2.e Behavior in the presence of a misuse of Unchecked_Deallocation, Unchecked_Access, or imported or exported entity (see Section 13).

3 [As explained in 1.1.3, “Conformity of an Implementation with the Standard”, the external effect of a program is defined in terms of its interactions with its external environment. Hence, the implementation can perform any internal actions whatsoever, in any order or in parallel, so long as the external effect of the execution of the program is one that is allowed by the canonical semantics, or by the rules of this clause.]

3.a **Ramification:** Note that an optimization can change the external effect of the program, so long as the changed external effect is an external effect that is allowed by the semantics. Note that the canonical semantics of an erroneous execution allows any external effect whatsoever. Hence, if the implementation can prove that program execution will be erroneous in certain circumstances, there need not be any constraints on the machine code executed in those circumstances.

Implementation Permissions

4 The following additional permissions are granted to the implementation:

- 5 {*extra permission to avoid raising exceptions*} {*undefined result*} An implementation need not always raise an exception when a language-defined check fails. Instead, the operation that failed the check can simply yield an *undefined result*. The exception need be raised by the implementation only if, in the absence of raising it, the value of this undefined result would have some effect on the external interactions of the program. In determining this, the implementation shall not presume that an undefined result has a value that belongs to its subtype, nor even to the base range of its type, if scalar. [Having removed the raise of the exception, the canonical semantics will in general allow the implementation to omit the code for the check, and some or all of the operation itself.]

Ramification: Even without this permission, an implementation can always remove a check if it cannot possibly fail. 5.a

Reason: We express the permission in terms of removing the raise, rather than the operation or the check, as it minimizes the disturbance to the canonical semantics (thereby simplifying reasoning). By allowing the implementation to omit the raise, it thereby does not need to "look" at what happens in the exception handler to decide whether the optimization is allowed. 5.b

Discussion: The implementation can also omit checks if they cannot possibly fail, or if they could only fail in erroneous executions. This follows from the canonical semantics. 5.c

Implementation Note: This permission is intended to allow normal "dead code removal" optimizations, even if some of the removed code might have failed some language-defined check. However, one may not eliminate the raise of an exception if subsequent code presumes in some way that the check succeeded. For example: 5.d

5.e
```
    if X * Y > Integer'Last then
        Put_Line("X * Y overflowed");
    end if;
  exception
    when others =>
        Put_Line("X * Y overflowed");
```

If X*Y does overflow, you may not remove the raise of the exception if the code that does the comparison against Integer'Last presumes that it is comparing it with an in-range Integer value, and hence always yields False. 5.e.1

As another example where a raise may not be eliminated: 5.f

5.g
```
    subtype Str10 is String(1..10);
    type P10 is access Str10;
    X : P10 := null;
  begin
    if X.all'Last = 10 then
        Put_Line("Oops");
    end if;
```

In the above code, it would be wrong to eliminate the raise of Constraint_Error on the "X.all" (since X is null), if the code to evaluate 'Last always yields 10 by presuming that X.all belongs to the subtype Str10, without even "looking." 5.g.1

- {*extra permission to reorder actions*} If an exception is raised due to the failure of a language-defined check, then upon reaching the corresponding exception_handler (or the termination of the task, if none), the external interactions that have occurred need reflect only that the exception was raised somewhere within the execution of the sequence_of_statements with the handler (or the task_body), possibly earlier (or later if the interactions are independent of the result of the checked operation) than that defined by the canonical semantics, but not within the execution of some abort-deferred operation or *independent* subprogram that does not dynamically enclose the execution of the construct whose check failed. {*independent subprogram*} An independent subprogram is one that is defined outside the library unit containing the construct whose check failed, and has no Inline pragma applied to it. {*normal state of an object*} {*abnormal state of an object* [partial]} {*disruption of an assignment* [partial]} Any assignment that occurred outside of such abort-deferred operations or independent subprograms can be disrupted by the raising of the exception, causing the object or its parts to become abnormal, and certain subsequent uses of the object to be erroneous, as explained in 13.9.1. 6

Reason: We allow such variables to become abnormal so that assignments (other than to atomic variables) can be disrupted due to “imprecise” exceptions or instruction scheduling, and so that assignments can be reordered so long as the correct results are produced in the end if no language-defined checks fail. 6.a

Ramification: If a check fails, no result dependent on the check may be incorporated in an external interaction. In other words, there is no permission to output meaningless results due to postponing a check. 6.b

Discussion: We believe it is important to state the extra permission to reorder actions in terms of what the programmer can expect at run time, rather than in terms of what the implementation can assume, or what transformations the implementation can perform. Otherwise, how can the programmer write reliable programs? 6.c

This clause has two conflicting goals: to allow as much optimization as possible, and to make program execution as predictable as possible (to ease the writing of reliable programs). The rules given above represent a compromise. 6.d

Consider the two extremes: 6.e

The extreme conservative rule would be to delete this clause entirely. The semantics of Ada would be the canonical semantics. This achieves the best predictability. It sounds like a disaster from the efficiency point of view, but in practice, implementations would provide modes in which less predictability but more efficiency would be achieved. 6.f

Such a mode could even be the out-of-the-box mode. In practice, implementers would provide a compromise based on their customer's needs. Therefore, we view this as one viable alternative.

6.g The extreme liberal rule would be “the language does not specify the execution of a program once a language-defined check has failed; such execution can be unpredictable.” This achieves the best efficiency. It sounds like a disaster from the predictability point of view, but in practice it might not be so bad. A user would have to assume that exception handlers for exceptions raised by language-defined checks are not portable. They would have to isolate such code (like all nonportable code), and would have to find out, for each implementation of interest, what behaviors can be expected. In practice, implementations would tend to avoid going so far as to punish their customers too much in terms of predictability.

6.h The most important thing about this clause is that users understand what they can expect at run time, and implementers understand what optimizations are allowed. Any solution that makes this clause contain rules that can interpreted in more than one way is unacceptable.

6.i We have chosen a compromise between the extreme conservative and extreme liberal rules. The current rule essentially allows arbitrary optimizations within a library unit and inlined subprograms reachable from it, but disallow semantics-disrupting optimizations across library units in the absence of inlined subprograms. This allows a library unit to be debugged, and then reused with some confidence that the abstraction it manages cannot be broken by bugs outside the library unit.

NOTES

7 5 The permissions granted by this clause can have an effect on the semantics of a program only if the program fails a language-defined check.

Wording Changes from Ada 83

7.a RM83-11.6 was unclear. It has been completely rewritten here; we hope this version is clearer. Here's what happened to each paragraph of RM83-11.6:

7.b • Paragraphs 1 and 2 contain no semantics; they are merely pointing out that anything goes if the canonical semantics is preserved. We have similar introductory paragraphs, but we have tried to clarify that these are not granting any “extra” permission beyond what the rest of the document allows.

7.c • Paragraphs 3 and 4 are reflected in the “extra permission to reorder actions”. Note that this permission now allows the reordering of assignments in many cases.

7.d • Paragraph 5 is moved to 4.5, “Operators and Expression Evaluation”, where operator association is discussed. Hence, this is no longer an “extra permission” but is part of the canonical semantics.

7.e • Paragraph 6 now follows from the general permission to store out-of-range values for unconstrained subtypes. Note that the parameters and results of all the predefined operators of a type are of the unconstrained subtype of the type.

7.f • Paragraph 7 is reflected in the “extra permission to avoid raising exceptions”.

7.g We moved clause 11.5, “Suppressing Checks” from after 11.6 to before 11.6, in order to preserve the famous number “11.6” (given the changes to earlier clauses in Section 11).

Section 12: Generic Units

{*generic unit*} A *generic unit* is a program unit that is either a generic subprogram or a generic package. {*template*} A generic unit is a *template*[, which can be parameterized, and from which corresponding (nongeneric) subprograms or packages can be obtained]. The resulting program units are said to be *instances* of the original generic unit. {*template: See generic unit*} {*macro: See generic unit*} {*parameter: See generic formal parameter*} 1

1.a **Glossary entry:** {*Generic unit*} A generic unit is a template for a (nongeneric) program unit; the template can be parameterized by objects, types, subprograms, and packages. An instance of a generic unit is created by a generic_instantiation. The rules of the language are enforced when a generic unit is compiled, using a generic contract model; additional checks are performed upon instantiation to verify the contract is met. That is, the declaration of a generic unit represents a contract between the body of the generic and instances of the generic. Generic units can be used to perform the role that macros sometimes play in other languages.

[A generic unit is declared by a generic_declaration. This form of declaration has a generic_formal_part declaring any generic formal parameters. An instance of a generic unit is obtained as the result of a generic_instantiation with appropriate generic actual parameters for the generic formal parameters. An instance of a generic subprogram is a subprogram. An instance of a generic package is a package. 2

Generic units are templates. As templates they do not have the properties that are specific to their nongeneric counterparts. For example, a generic subprogram can be instantiated but it cannot be called. In contrast, an instance of a generic subprogram is a (nongeneric) subprogram; hence, this instance can be called but it cannot be used to produce further instances.] 3

12.1 Generic Declarations

[A generic_declaration declares a generic unit, which is either a generic subprogram or a generic package. A generic_declaration includes a generic_formal_part declaring any generic formal parameters. A generic formal parameter can be an object; alternatively (unlike a parameter of a subprogram), it can be a type, a subprogram, or a package.] 1

Syntax

generic_declaration ::= generic_subprogram_declaration | generic_package_declaration 2

generic_subprogram_declaration ::= 3
    generic_formal_part subprogram_specification;

generic_package_declaration ::= 4
    generic_formal_part package_specification;

generic_formal_part ::= **generic** {generic_formal_parameter_declaration | use_clause} 5

generic_formal_parameter_declaration ::= 6
      formal_object_declaration
    | formal_type_declaration
    | formal_subprogram_declaration
    | formal_package_declaration

The only form of subtype_indication allowed within a generic_formal_part is a subtype_mark [(that is, the subtype_indication shall not include an explicit constraint)]. The defining name of a generic subprogram shall be an identifier [(not an operator_symbol)]. 7

7.a **Reason:** The reason for forbidding constraints in subtype_indications is that it simplifies the elaboration of generic_declarations (since there is nothing to evaluate), and that it simplifies the matching rules, and makes them more checkable at compile time.

Static Semantics

8/2 {*AI95-00434-01*} {*generic package*} {*generic subprogram*} {*generic procedure*} {*generic function*} A generic_declaration declares a generic unit — a generic package, generic procedure, or generic function, as appropriate.

9 {*generic formal*} An entity is a *generic formal* entity if it is declared by a generic_formal_parameter_declaration. "Generic formal," or simply "formal," is used as a prefix in referring to objects, subtypes (and types), functions, procedures and packages, that are generic formal entities, as well as to their respective declarations. [Examples: "generic formal procedure" or a "formal integer type declaration."]

Dynamic Semantics

10 {*elaboration (generic_declaration)* [partial]} The elaboration of a generic_declaration has no effect.

NOTES

11 1 Outside a generic unit a name that denotes the generic_declaration denotes the generic unit. In contrast, within the declarative region of the generic unit, a name that denotes the generic_declaration denotes the current instance.

11.a **Proof:** This is stated officially as part of the "current instance" rule in 8.6, "The Context of Overload Resolution". See also 12.3, "Generic Instantiation".

12 2 Within a generic subprogram_body, the name of this program unit acts as the name of a subprogram. Hence this name can be overloaded, and it can appear in a recursive call of the current instance. For the same reason, this name cannot appear after the reserved word **new** in a (recursive) generic_instantiation.

13 3 A default_expression or default_name appearing in a generic_formal_part is not evaluated during elaboration of the generic_formal_part; instead, it is evaluated when used. (The usual visibility rules apply to any name used in a default: the denoted declaration therefore has to be visible at the place of the expression.)

Examples

14 *Examples of generic formal parts:*

15
```
generic     --  parameterless
```

16
```
generic
   Size : Natural;  --  formal object
```

17
```
generic
   Length : Integer := 200;          -- formal object with a default expression
```

18
```
   Area   : Integer := Length*Length; -- formal object with a default expression
```

19
```
generic
   type Item  is private;                       -- formal type
   type Index is (<>);                          -- formal type
   type Row   is array(Index range <>) of Item; -- formal type
   with function "<"(X, Y : Item) return Boolean;    -- formal subprogram
```

20 *Examples of generic declarations declaring generic subprograms Exchange and Squaring:*

21
```
generic
   type Elem is private;
procedure Exchange(U, V : in out Elem);
```

22
```
generic
   type Item is private;
   with function "*"(U, V : Item) return Item is <>;
function Squaring(X : Item) return Item;
```

Example of a generic declaration declaring a generic package: 23

24
```
generic
   type Item   is private;
   type Vector is array (Positive range <>) of Item;
   with function Sum(X, Y : Item) return Item;
package On_Vectors is
   function Sum  (A, B : Vector) return Vector;
   function Sigma(A    : Vector) return Item;
   Length_Error : exception;
end On_Vectors;
```

Extensions to Ada 83

24.a {*extensions to Ada 83*} The syntax rule for generic_formal_parameter_declaration is modified to allow the reserved words **tagged** and **abstract**, to allow formal derived types, and to allow formal packages.

24.b Use_clauses are allowed in generic_formal_parts. This is necessary in order to allow a use_clause within a formal part to provide direct visibility of declarations within a generic formal package.

Wording Changes from Ada 83

24.c The syntax for generic_formal_parameter_declaration and formal_type_definition is split up into more named categories. The rules for these categories are moved to the appropriate clauses and subclauses. The names of the categories are changed to be more intuitive and uniform. For example, we changed generic_parameter_declaration to generic_formal_parameter_declaration, because the thing it declares is a generic formal, not a generic. In the others, we abbreviate “generic_formal” to just “formal”. We can't do that for generic_formal_parameter_declaration, because of confusion with normal formal parameters of subprograms.

12.2 Generic Bodies

1 {*generic body*} The body of a generic unit (a *generic body*) [is a template for the instance bodies. The syntax of a generic body is identical to that of a nongeneric body].

1.a **Ramification:** We also use terms like “generic function body” and “nongeneric package body.”

Dynamic Semantics

2 {*elaboration (generic body)* [partial]} The elaboration of a generic body has no other effect than to establish that the generic unit can from then on be instantiated without failing the Elaboration_Check. If the generic body is a child of a generic package, then its elaboration establishes that each corresponding declaration nested in an instance of the parent (see 10.1.1) can from then on be instantiated without failing the Elaboration_Check.

NOTES

3 4 The syntax of generic subprograms implies that a generic subprogram body is always the completion of a declaration.

Examples

4 *Example of a generic procedure body:*

5
```
procedure Exchange(U, V : in out Elem) is  -- see 12.1
   T : Elem;  -- the generic formal type
begin
   T := U;
   U := V;
   V := T;
end Exchange;
```

6 *Example of a generic function body:*

7
```
function Squaring(X : Item) return Item is  -- see 12.1
begin
   return X*X;  -- the formal operator "*"
end Squaring;
```

8 *Example of a generic package body:*

9
```
package body On_Vectors is  -- see 12.1
```
10
```
   function Sum(A, B : Vector) return Vector is
      Result : Vector(A'Range); -- the formal type Vector
      Bias   : constant Integer := B'First - A'First;
   begin
      if A'Length /= B'Length then
         raise Length_Error;
      end if;
```
11
```
      for N in A'Range loop
         Result(N) := Sum(A(N), B(N + Bias)); -- the formal function Sum
      end loop;
      return Result;
   end Sum;
```
12
```
   function Sigma(A : Vector) return Item is
      Total : Item := A(A'First); -- the formal type Item
   begin
      for N in A'First + 1 .. A'Last loop
         Total := Sum(Total, A(N)); -- the formal function Sum
      end loop;
      return Total;
   end Sigma;
end On_Vectors;
```

12.3 Generic Instantiation

1 [{*instance (of a generic unit)*} An instance of a generic unit is declared by a generic_instantiation.]

Language Design Principles

1.a {*generic contract model*} {*contract model of generics*} The legality of an instance should be determinable without looking at the generic body. Likewise, the legality of a generic body should be determinable without looking at any instances. Thus, the generic_declaration forms a contract between the body and the instances; if each obeys the rules with respect to the generic_declaration, then no legality problems will arise. This is really a special case of the “legality determinable via semantic dependences” Language Design Principle (see Section 10), given that a generic_instantiation does not depend semantically upon the generic body, nor vice-versa.

1.b Run-time issues are another story. For example, whether parameter passing is by copy or by reference is determined in part by the properties of the generic actuals, and thus cannot be determined at compile time of the generic body. Similarly, the contract model does not apply to Post-Compilation Rules.

Syntax

2/2 {*AI95-00218-03*} generic_instantiation ::=
  package defining_program_unit_name **is**
    new *generic_package*_name [generic_actual_part];
 | [overriding_indicator]
  procedure defining_program_unit_name **is**
    new *generic_procedure*_name [generic_actual_part];
 | [overriding_indicator]
  function defining_designator **is**
    new *generic_function*_name [generic_actual_part];

3 generic_actual_part ::=
  (generic_association {, generic_association})

4 generic_association ::=
  [*generic_formal_parameter*_selector_name =>] explicit_generic_actual_parameter

explicit_generic_actual_parameter ::= expression | *variable*_name
  | *subprogram*_name | *entry*_name | subtype_mark
  | *package_instance*_name 5

{*named association*} {*positional association*} A generic_association is *named* or *positional* according to whether or not the *generic_formal_parameter*_selector_name is specified. Any positional associations shall precede any named associations. 6

{*generic actual parameter*} {*generic actual*} {*actual*} The *generic actual parameter* is either the explicit_generic_actual_parameter given in a generic_association for each formal, or the corresponding default_expression or default_name if no generic_association is given for the formal. When the meaning is clear from context, the term "generic actual," or simply "actual," is used as a synonym for "generic actual parameter" and also for the view denoted by one, or the value of one. 7/2

Legality Rules

In a generic_instantiation for a particular kind of program unit [(package, procedure, or function)], the name shall denote a generic unit of the corresponding kind [(generic package, generic procedure, or generic function, respectively)]. 8

The *generic_formal_parameter*_selector_name of a generic_association shall denote a generic_formal_parameter_declaration of the generic unit being instantiated. If two or more formal subprograms have the same defining name, then named associations are not allowed for the corresponding actuals. 9

A generic_instantiation shall contain at most one generic_association for each formal. Each formal without an association shall have a default_expression or subprogram_default. 10

In a generic unit Legality Rules are enforced at compile time of the generic_declaration and generic body, given the properties of the formals. In the visible part and formal part of an instance, Legality Rules are enforced at compile time of the generic_instantiation, given the properties of the actuals. In other parts of an instance, Legality Rules are not enforced; this rule does not apply when a given rule explicitly specifies otherwise. 11

Reason: {*AI95-00114-01*} Since rules are checked using the properties of the formals, and since these properties do not always carry over to the actuals, we need to check the rules again in the visible part of the instance. For example, only if a tagged type is limited may an extension of it have limited components in the record_extension_part. A formal tagged limited type is limited, but the actual might be nonlimited. Hence any rule that requires a tagged type to be limited runs into this problem. Such rules are rare; in most cases, the rules for matching of formals and actuals guarantee that if the rule is obeyed in the generic unit, then it has to be obeyed in the instance. 11.a/2

Ramification: The "properties" of the formals are determined without knowing anything about the actuals: 11.b

- {*8652/0095*} {*AI95-00034-01*} A formal derived subtype is constrained if and only if the ancestor subtype is constrained. A formal array type is constrained if and only if the declarations say so. A formal private type is constrained if it does not have a discriminant part. Other formal subtypes are unconstrained, even though they might be constrained in an instance. 11.c/1
- A formal subtype can be indefinite, even though the copy might be definite in an instance. 11.d
- A formal object of mode **in** is not a static constant; in an instance, the copy is static if the actual is. 11.e
- A formal subtype is not static, even though the actual might be. 11.f
- Formal types are specific, even though the actual can be class-wide. 11.g
- The subtype of a formal object of mode **in out** is not static. (This covers the case of AI83-00878.) 11.h
- The subtype of a formal parameter of a formal subprogram does not provide an applicable index constraint. 11.i
- The profile of a formal subprogram is not subtype-conformant with any other profile. {*subtype conformance*} 11.j
- A generic formal function is not static. 11.k

Ramification: The exceptions to the above rule about when legality rules are enforced fall into these categories: 11.l

- Some rules are checked in the generic declaration, and then again in both the visible and private parts of the instance: 11.m

11.n
- The parent type of a record extension has to be specific (see 3.9.1). This rule is not checked in the instance body.

11.o
- The parent type of a private extension has to be specific (see 7.3). This rule is not checked in the instance body.

11.p/2
- {*AI95-00402-01*} A type with a default_expression of an access discriminant has to be a descendant of an explicitly limited record type, or be a task or protected type. This rule is irrelevant in the instance body.]}

11.q
- In the declaration of a record extension, if the parent type is nonlimited, then each of the components of the record_extension_part have to be nonlimited (see 3.9.1). In the generic body, this rule is checked in an assume-the-worst manner.

11.r
- A preelaborated library unit has to be preelaborable (see 10.2.1). In the generic body, this rule is checked in an assume-the-worst manner.

11.r.1/2 {*AI95-00402-01*} The corrections made by the Corrigendum added a number of such rules, and the Amendment added many more. There doesn't seem to be much value in repeating all of these rules here (as of this writing, there are roughly 17 such rules). As noted below, all such rules are indexed in the AARM.

11.s
- {*accessibility rule (checking in generic units)* [partial]} For the accessibility rules, the formals have nothing to say about the property in question. Like the above rules, these rules are checked in the generic declaration, and then again in both the visible and private parts of the instance. In the generic body, we have explicit rules that essentially assume the worst (in the cases of type extensions and access-to-subprogram types), and we have run-time checks (in the case of access-to-object types). See 3.9.1, 3.10.2, and 4.6.

11.t We considered run-time checks for access-to-subprogram types as well. However, this would present difficulties for implementations that share generic bodies.

11.u
- The rules requiring “reasonable” values for static expressions are ignored when the expected type for the expression is a descendant of a generic formal type other than a generic formal derived type, and do not apply in an instance.

11.v
- The rule forbidding two explicit homographs in the same declarative region does not apply in an instance of a generic unit, except that it *does* apply in the declaration of a record extension that appears in the visible part of an instance.

11.w
- Some rules do not apply at all in an instance, not even in the visible part:

11.x
  - Body_stubs are not normally allowed to be multiply nested, but they can be in instances.

11.y {*generic contract issue* [distributed]} Each rule that is an exception is marked with “generic contract issue;” look that up in the index to find them all.

11.z **Ramification:** The Legality Rules are the ones labeled Legality Rules. We are talking about all Legality Rules in the entire language here. Note that, with some exceptions, the legality of a generic unit is checked even if there are no instantiations of the generic unit.

11.aa **Ramification:** The Legality Rules are described here, and the overloading rules were described earlier in this clause. Presumably, every Static Semantic Item is sucked in by one of those. Thus, we have covered all the compile-time rules of the language. There is no need to say anything special about the Post-Compilation Rules or the Dynamic Semantic Items.

11.bb **Discussion:** Here is an example illustrating how this rule is checked: “In the declaration of a record extension, if the parent type is nonlimited, then each of the components of the record_extension_part shall be nonlimited.”

11.cc
```
generic
    type Parent is tagged private;
    type Comp is limited private;
package G1 is
    type Extension is new Parent with
        record
            C : Comp; -- Illegal!
        end record;
end G1;
```

11.dd/1 The parent type is nonlimited, and the component type is limited, which is illegal. It doesn't matter that one could imagine writing an instantiation with the actual for Comp being nonlimited — we never get to the instance, because the generic itself is illegal.

11.ee On the other hand:

11.ff
```
generic
    type Parent is tagged limited private; -- Parent is limited.
    type Comp is limited private;
package G2 is
    type Extension is new Parent with
        record
            C : Comp; -- OK.
        end record;
end G2;
```

11.gg
```
type Limited_Tagged is tagged limited null record;
type Non_Limited_Tagged is tagged null record;
```

11.hh
```
type Limited_Untagged is limited null record;
type Non_Limited_Untagged is null record;
```

11.ii
```
package Good_1 is new G2(Parent => Limited_Tagged,
                         Comp => Limited_Untagged);
package Good_2 is new G2(Parent => Non_Limited_Tagged,
                         Comp => Non_Limited_Untagged);
package Bad  is new G2(Parent => Non_Limited_Tagged,
                         Comp => Limited_Untagged); -- Illegal!
```

11.jj The first instantiation is legal, because in the instance the parent is limited, so the rule is not violated. Likewise, in the second instantiation, the rule is not violated in the instance. However, in the Bad instance, the parent type is nonlimited, and the component type is limited, so this instantiation is illegal.

Static Semantics

12 A generic_instantiation declares an instance; it is equivalent to the instance declaration (a package_-declaration or subprogram_declaration) immediately followed by the instance body, both at the place of the instantiation.

12.a **Ramification:** The declaration and the body of the instance are not “implicit” in the technical sense, even though you can't see them in the program text. Nor are declarations within an instance “implicit” (unless they are implicit by other rules). This is necessary because implicit declarations have special semantics that should not be attached to instances. For a generic subprogram, the profile of a generic_instantiation is that of the instance declaration, by the stated equivalence.

12.b **Ramification:** {*visible part (of an instance)* [partial]} {*private part (of a package)* [partial]} The visible and private parts of a package instance are defined in 7.1, “Package Specifications and Declarations” and 12.7, “Formal Packages”. The visible and private parts of a subprogram instance are defined in 8.2, “Scope of Declarations”.

13 The instance is a copy of the text of the template. [Each use of a formal parameter becomes (in the copy) a use of the actual, as explained below.] {*package instance*} {*subprogram instance*} {*procedure instance*} {*function instance*} {*instance (of a generic package)*} {*instance (of a generic subprogram)*} {*instance (of a generic procedure)*} {*instance (of a generic function)*} An instance of a generic package is a package, that of a generic procedure is a procedure, and that of a generic function is a function.

13.a **Ramification:** An instance is a package or subprogram (because we say so), even though it contains a copy of the generic_formal_part, and therefore doesn't look like one. This is strange, but it's OK, since the syntax rules are overloading rules, and therefore do not apply in an instance.

13.b **Discussion:** We use a macro-expansion model, with some explicitly-stated exceptions (see below). The main exception is that the interpretation of each construct in a generic unit (especially including the denotation of each name) is determined when the declaration and body of the generic unit (as opposed to the instance) are compiled, and in each instance this interpretation is (a copy of) the template interpretation. In other words, if a construct is interpreted as a name denoting a declaration D, then in an instance, the copy of the construct will still be a name, and will still denote D (or a copy of D). From an implementation point of view, overload resolution is performed on the template, and not on each copy.

13.c We describe the substitution of generic actual parameters by saying (in most cases) that the copy of each generic formal parameter declares a view of the actual. Suppose a name in a generic unit denotes a generic_formal_parameter_declaration. The copy of that name in an instance will denote the copy of that generic_formal_parameter_declaration in the instance. Since the generic_formal_parameter_declaration in the instance declares a view of the actual, the name will denote a view of the actual.

13.d/2 {*AI95-00442-01*} Other properties of the copy (for example, staticness, categories to which types belong) are recalculated for each instance; this is implied by the fact that it's a copy.

13.e/2 {*AI95-00317-01*} Although the generic_formal_part is included in an instance, the declarations in the generic_formal_part are only visible outside the instance in the case of a generic formal package whose formal_package_actual_part includes one or more <> indicators — see 12.7.

14 The interpretation of each construct within a generic declaration or body is determined using the overloading rules when that generic declaration or body is compiled. In an instance, the interpretation of each (copied) construct is the same, except in the case of a name that denotes the generic_declaration or some declaration within the generic unit; the corresponding name in the instance then denotes the corresponding copy of the denoted declaration. The overloading rules do not apply in the instance.

14.a **Ramification:** See 8.6, "The Context of Overload Resolution" for definitions of "interpretation" and "overloading rule."

14.b Even the generic_formal_parameter_declarations have corresponding declarations in the instance, which declare views of the actuals.

14.c Although the declarations in the instance are copies of those in the generic unit, they often have quite different properties, as explained below. For example a constant declaration in the generic unit might declare a nonstatic constant, whereas the copy of that declaration might declare a static constant. This can happen when the staticness depends on some generic formal.

14.d This rule is partly a ramification of the "current instance" rule in 8.6, "The Context of Overload Resolution". Note that that rule doesn't cover the generic_formal_part.

14.e Although the overloading rules are not observed in the instance, they are, of course, observed in the _instantiation in order to determine the interpretation of the constituents of the _instantiation.

14.f Since children are considered to occur within their parent's declarative region, the above rule applies to a name that denotes a child of a generic unit, or a declaration inside such a child.

14.g Since the Syntax Rules are overloading rules, it is possible (legal) to violate them in an instance. For example, it is possible for an instance body to occur in a package_specification, even though the Syntax Rules forbid bodies in package_specifications.

15 In an instance, a generic_formal_parameter_declaration declares a view whose properties are identical to those of the actual, except as specified in 12.4, "Formal Objects" and 12.6, "Formal Subprograms". Similarly, for a declaration within a generic_formal_parameter_declaration, the corresponding declaration in an instance declares a view whose properties are identical to the corresponding declaration within the declaration of the actual.

15.a **Ramification:** In an instance, there are no "properties" of types and subtypes that come from the formal. The primitive operations of the type come from the formal, but these are declarations in their own right, and are therefore handled separately.

15.b Note that certain properties that come from the actuals are irrelevant in the instance. For example, if an actual type is of a class deeper in the derived-type hierarchy than the formal, it is impossible to call the additional operations of the deeper class in the instance, because any such call would have to be a copy of some corresponding call in the generic unit, which would have been illegal. However, it is sometimes possible to reach into the specification of the instance from outside, and notice such properties. For example, one could pass an object declared in the instance specification to one of the additional operations of the deeper type.

15.c/2 {*AI95-00114-01*} A formal_type_declaration can contain discriminant_specifications, a formal_subprogram_declaration can contain parameter_specifications, and a formal_package_declaration can contain many kinds of declarations. These are all inside the generic unit, and have corresponding declarations in the instance.

15.d This rule implies, for example, that if a subtype in a generic unit is a subtype of a generic formal subtype, then the corresponding subtype in the instance is a subtype of the corresponding actual subtype.

15.e For a generic_instantiation, if a generic actual is a static [(scalar or string)] subtype, then each use of the corresponding formal parameter within the specification of the instance is considered to be static. (See AI83-00409.)

15.f Similarly, if a generic actual is a static expression and the corresponding formal parameter has a static [(scalar or string)] subtype, then each use of the formal parameter in the specification of the instance is considered to be static. (See AI83-00505.)

15.g If a primitive subprogram of a type derived from a generic formal derived tagged type is not overriding (that is, it is a new subprogram), it is possible for the copy of that subprogram in an instance to override a subprogram inherited from the actual. For example:

```
type T1 is tagged record ... end record;                                   15.h

generic                                                                    15.i
    type Formal is new T1;
package G is
    type Derived_From_Formal is new Formal with record ... end record;
    procedure Foo(X : in Derived_From_Formal); -- Does not override anything.
end G;

type T2 is new T1 with record ... end record;                              15.j
procedure Foo(X : in T2);

package Inst is new G(Formal => T2);                                       15.k
```

In the instance Inst, the declaration of Foo for Derived_From_Formal overrides the Foo inherited from T2. 15.l

Implementation Note: {*8652/0009*} {*AI95-00137-01*} For formal types, an implementation that shares the code among multiple instances of the same generic unit needs to beware that things like parameter passing mechanisms (by-copy vs. by-reference) and aspect_clauses are determined by the actual. 15.m/1

16 [Implicit declarations are also copied, and a name that denotes an implicit declaration in the generic denotes the corresponding copy in the instance. However, for a type declared within the visible part of the generic, a whole new set of primitive subprograms is implicitly declared for use outside the instance, and may differ from the copied set if the properties of the type in some way depend on the properties of some actual type specified in the instantiation. For example, if the type in the generic is derived from a formal private type, then in the instance the type will inherit subprograms from the corresponding actual type.

17 {*override*} These new implicit declarations occur immediately after the type declaration in the instance, and override the copied ones. The copied ones can be called only from within the instance; the new ones can be called only from outside the instance, although for tagged types, the body of a new one can be executed by a call to an old one.]

Proof: This rule is stated officially in 8.3, “Visibility”. 17.a

Ramification: The new ones follow from the class(es) of the formal types. For example, for a type T derived from a generic formal private type, if the actual is Integer, then the copy of T in the instance has a "+" primitive operator, which can be called from outside the instance (assuming T is declared in the visible part of the instance). 17.b

AI83-00398. 17.c

{*AI95-00442-01*} Since an actual type is always in the category determined for the formal, the new subprograms hide all of the copied ones, except for a declaration of "/=" that corresponds to an explicit declaration of "=". Such "/=" operators are special, because unlike other implicit declarations of primitive subprograms, they do not appear by virtue of the class, but because of an explicit declaration of "=". If the declaration of "=" is implicit (and therefore overridden in the instance), then a corresponding implicitly declared "/=" is also overridden. But if the declaration of "=" is explicit (and therefore not overridden in the instance), then a corresponding implicitly declared "/=" is not overridden either, even though it's implicit. 17.d/2

Note that the copied ones can be called from inside the instance, even though they are hidden from all visibility, because the names are resolved in the generic unit — visibility is irrelevant for calls in the instance. 17.e

18 [In the visible part of an instance, an explicit declaration overrides an implicit declaration if they are homographs, as described in 8.3.] On the other hand, an explicit declaration in the private part of an instance overrides an implicit declaration in the instance, only if the corresponding explicit declaration in the generic overrides a corresponding implicit declaration in the generic. Corresponding rules apply to the other kinds of overriding described in 8.3.

Ramification: For example: 18.a

```
type Ancestor is tagged null record;                                       18.b

generic                                                                    18.c
    type Formal is new Ancestor with private;
package G is
    type T is new Formal with null record;
    procedure P(X : in T); -- (1)
private
    procedure Q(X : in T); -- (2)
end G;
```

18.d
```
        type Actual is new Ancestor with null record;
        procedure P(X : in Actual);
        procedure Q(X : in Actual);
```
18.e
```
        package Instance is new G(Formal => Actual);
```
18.f In the instance, the copy of P at (1) overrides Actual's P, whereas the copy of Q at (2) does not override anything; in implementation terms, it occupies a separate slot in the type descriptor.

18.g **Reason:** The reason for this rule is so a programmer writing an _instantiation need not look at the private part of the generic in order to determine which subprograms will be overridden.

Post-Compilation Rules

19 Recursive generic instantiation is not allowed in the following sense: if a given generic unit includes an instantiation of a second generic unit, then the instance generated by this instantiation shall not include an instance of the first generic unit [(whether this instance is generated directly, or indirectly by intermediate instantiations)].

19.a **Discussion:** Note that this rule is not a violation of the generic contract model, because it is not a Legality Rule. Some implementations may be able to check this rule at compile time, but that requires access to all the bodies, so we allow implementations to check the rule at link time.

Dynamic Semantics

20 {*elaboration (generic_instantiation)* [partial]} For the elaboration of a generic_instantiation, each generic_association is first evaluated. If a default is used, an implicit generic_association is assumed for this rule. These evaluations are done in an arbitrary order, except that the evaluation for a default actual takes place after the evaluation for another actual if the default includes a name that denotes the other one. Finally, the instance declaration and body are elaborated.

20.a **Ramification:** Note that if the evaluation of a default depends on some side-effect of some other evaluation, the order is still arbitrary.

21 {*evaluation (generic_association)* [partial]} For the evaluation of a generic_association the generic actual parameter is evaluated. Additional actions are performed in the case of a formal object of mode **in** (see 12.4).

21.a **To be honest:** Actually, the actual is evaluated only if evaluation is defined for that kind of construct — we don't actually “evaluate” subtype_marks.

NOTES

22 5 If a formal type is not tagged, then the type is treated as an untagged type within the generic body. Deriving from such a type in a generic body is permitted; the new type does not get a new tag value, even if the actual is tagged. Overriding operations for such a derived type cannot be dispatched to from outside the instance.

22.a **Ramification:** If two overloaded subprograms declared in a generic package specification differ only by the (formal) type of their parameters and results, then there exist legal instantiations for which all calls of these subprograms from outside the instance are ambiguous. For example:

22.b
```
        generic
           type A is (<>);
           type B is private;
        package G is
           function Next(X : A) return A;
           function Next(X : B) return B;
        end G;
```
22.c
```
        package P is new G(A => Boolean, B => Boolean);
        -- All calls of P.Next are ambiguous.
```
22.d **Ramification:** The following example illustrates some of the subtleties of the substitution of formals and actuals:

22.e
```
generic
    type T1 is private;
    -- A predefined "=" operator is implicitly declared here:
    -- function "="(Left, Right : T1) return Boolean;
    -- Call this "="1.
package G is
    subtype S1 is T1; -- So we can get our hands on the type from
                      -- outside an instance.
    type T2 is new T1;
    -- An inherited "=" operator is implicitly declared here:
    -- function "="(Left, Right : T2) return Boolean;
    -- Call this "="2.
```

22.f
```
    T1_Obj : T1 := ...;
    Bool_1 : Boolean := T1_Obj = T1_Obj;
```

22.g
```
    T2_Obj : T2 := ...;
    Bool_2 : Boolean := T2_Obj = T2_Obj;
end G;
...
```

22.h
```
package P is
    type My_Int is new Integer;
    -- A predefined "=" operator is implicitly declared here:
    -- function "="(Left, Right : My_Int) return Boolean;
    -- Call this "="3.
    function "="(X, Y : My_Int) return Boolean;
    -- Call this "="4.
    -- "="3 is hidden from all visibility by "="4.
    -- Nonetheless, "="3 can “reemerge” in certain circumstances.
end P;
use P;
...
package I is new G(T1 => My_Int); -- "="5 is declared in I (see below).
use I;
```

22.i
```
Another_T1_Obj : S1 := 13; -- Can't denote T1, but S1 will do.
Bool_3 : Boolean := Another_T1_Obj = Another_T1_Obj;
```

22.j
```
Another_T2_Obj : T2 := 45;
Bool_4 : Boolean := Another_T2_Obj = Another_T2_Obj;
```

22.k
```
Double : T2 := T2_Obj + Another_T2_Obj;
```

22.l In the instance I, there is a copy of "="$_1$ (call it "="$_{1i}$) and "="$_2$ (call it "="$_{2i}$). The "="$_{1i}$ and "="$_{2i}$ declare views of the predefined "=" of My_Int (that is, "="$_3$). In the initialization of Bool_1 and Bool_2 in the generic unit G, the names "=" denote "="$_1$ and "="$_2$, respectively. Therefore, the copies of these names in the instances denote "="$_{1i}$ and "="$_{2i}$, respectively. Thus, the initialization of I.Bool_1 and I.Bool_2 call the predefined equality operator of My_Int; they will not call "="$_4$.

22.m The declarations "="$_{1i}$ and "="$_{2i}$ are hidden from all visibility. This prevents them from being called from outside the instance.

22.n The declaration of Bool_3 calls "="$_4$.

22.o The instance I also contains implicit declarations of the primitive operators of T2, such as "=" (call it "="$_5$) and "+". These operations cannot be called from within the instance, but the declaration of Bool_4 calls "="$_5$.

Examples

23 *Examples of generic instantiations (see 12.1):*

24
```
procedure Swap is new Exchange(Elem => Integer);
procedure Swap is new Exchange(Character);          -- Swap is overloaded
function Square is new Squaring(Integer);   -- "*" of Integer used by default
function Square is new Squaring(Item => Matrix, "*" => Matrix_Product);
function Square is new Squaring(Matrix, Matrix_Product); -- same as previous
```

25
```
package Int_Vectors is new On_Vectors(Integer, Table, "+");
```

26 *Examples of uses of instantiated units:*

27
```
Swap(A, B);
A := Square(A);
```

28
```
T : Table(1 .. 5) := (10, 20, 30, 40, 50);
N : Integer := Int_Vectors.Sigma(T);  -- 150 (see 12.2, “Generic Bodies” for the body of Sigma)
```

29
```
    use Int_Vectors;
    M : Integer := Sigma(T);  -- 150
```

Inconsistencies With Ada 83

29.a {*inconsistencies with Ada 83*} In Ada 83, all explicit actuals are evaluated before all defaults, and the defaults are evaluated in the order of the formal declarations. This ordering requirement is relaxed in Ada 95.

Incompatibilities With Ada 83

29.b {*incompatibilities with Ada 83*} We have attempted to remove every violation of the contract model. Any remaining contract model violations should be considered bugs in the RM95. The unfortunate property of reverting to the predefined operators of the actual types is retained for upward compatibility. (Note that fixing this would require subtype conformance rules.) However, tagged types do not revert in this sense.

Extensions to Ada 83

29.c {*extensions to Ada 83*} The syntax rule for explicit_generic_actual_parameter is modified to allow a *package_instance_*name.

Wording Changes from Ada 83

29.d The fact that named associations cannot be used for two formal subprograms with the same defining name is moved to AARM-only material, because it is a ramification of other rules, and because it is not of interest to the average user.

29.e/2 {*AI95-00114-01*} The rule that “An explicit explicit_generic_actual_parameter shall not be supplied more than once for a given generic formal parameter” seems to be missing from RM83, although it was clearly the intent.

29.f In the explanation that the instance is a copy of the template, we have left out RM83-12.3(5)'s “apart from the generic formal part”, because it seems that things in the formal part still need to exist in instances. This is particularly true for generic formal packages, where you're sometimes allowed to reach in and denote the formals of the formal package from outside it. This simplifies the explanation of what each name in an instance denotes: there are just two cases: the declaration can be inside or outside (where inside needs to include the generic unit itself). Note that the RM83 approach of listing many cases (see RM83-12.5(5-14)) would have become even more unwieldy with the addition of generic formal packages, and the declarations that occur therein.

29.g We have corrected the definition of the elaboration of a generic_instantiation (RM83-12.3(17)); we don't elaborate entities, and the instance is not “implicit.”

29.h In RM83, there is a rule saying the formal and actual shall match, and then there is much text defining what it means to match. Here, we simply state all the latter text as rules. For example, “A formal foo is matched by an actual greenish bar” becomes “For a formal foo, the actual shall be a greenish bar.” This is necessary to split the Name Resolution Rules from the Legality Rules. Besides, there's really no need to define the concept of matching for generic parameters.

Extensions to Ada 95

29.i/2 {*AI95-00218-03*} {*extensions to Ada 95*} An overriding_indicator (see 8.3.1) is allowed on a subprogram instantiation.

12.4 Formal Objects

1 [{*generic formal object*} {*formal object, generic*} A generic formal object can be used to pass a value or variable to a generic unit.]

Language Design Principles

1.a A generic formal object of mode **in** is like a constant initialized to the value of the explicit_generic_actual_parameter.

1.b A generic formal object of mode **in out** is like a renaming of the explicit_generic_actual_parameter.

Syntax

2/2 {*AI95-00423-01*} formal_object_declaration ::=
  defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression];
  defining_identifier_list : mode access_definition [:= default_expression];

Name Resolution Rules

3 {*expected type (generic formal object default_expression)* [partial]} The expected type for the default_expression, if any, of a formal object is the type of the formal object.

4 {*expected type (generic formal in object actual)* [partial]} For a generic formal object of mode **in**, the expected type for the actual is the type of the formal.

5/2 {*AI95-00423-01*} For a generic formal object of mode **in out**, the type of the actual shall resolve to the type determined by the subtype_mark, or for a formal_object_declaration with an access_definition, to a specific anonymous access type. If the anonymous access type is an access-to-object type, the type of the actual shall have the same designated type as that of the access_definition. If the anonymous access type is an access-to-subprogram type, the type of the actual shall have a designated profile which is type conformant with that of the access_definition. {*type conformance (required)*} .

5.a **Reason:** See the corresponding rule for object_renaming_declarations for a discussion of the reason for this rule.

Legality Rules

6 If a generic formal object has a default_expression, then the mode shall be **in** [(either explicitly or by default)]; otherwise, its mode shall be either **in** or **in out**.

6.a **Ramification:** Mode **out** is not allowed for generic formal objects.

7 For a generic formal object of mode **in**, the actual shall be an expression. For a generic formal object of mode **in out**, the actual shall be a name that denotes a variable for which renaming is allowed (see 8.5.1).

7.a **To be honest:** The part of this that requires an expression or name is a Name Resolution Rule, but that's too pedantic to worry about. (The part about denoting a variable, and renaming being allowed, is most certainly *not* a Name Resolution Rule.)

8/2 {*AI95-00287-01*} {*AI95-00423-01*} In the case where the type of the formal is defined by an access_definition, the type of the actual and the type of the formal:

- 8.1/2 {*AI95-00423-01*} shall both be access-to-object types with statically matching designated subtypes and with both or neither being access-to-constant types; or {*statically matching (required)* [partial]}
- 8.2/2 {*AI95-00423-01*} shall both be access-to-subprogram types with subtype conformant designated profiles. {*subtype conformance (required)*}

8.3/2 {*AI95-00423-01*} For a formal_object_declaration with a null_exclusion or an access_definition that has a null_exclusion:

- 8.4/2 if the actual matching the formal_object_declaration denotes the generic formal object of another generic unit *G*, and the instantiation containing the actual occurs within the body of *G* or within the body of a generic unit declared within the declarative region of *G*, then the declaration of the formal object of *G* shall have a null_exclusion;
- 8.5/2 otherwise, the subtype of the actual matching the formal_object_declaration shall exclude null. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

8.a/2 **Reason:** {*AI95-00287-01*} {*AI95-00423-01*} This rule prevents “lying”. **Null** must never be the value of an object with an explicit null_exclusion. The first bullet is an assume-the-worst rule which prevents trouble in generic bodies (including bodies of child units) when the subtype of the formal object excludes null implicitly.

Static Semantics

9/2 {*AI95-00255-01*} {*AI95-00423-01*} A formal_object_declaration declares a generic formal object. The default mode is **in**. {*nominal subtype (of a generic formal object)* [partial]} For a formal object of mode **in**, the nominal subtype is the one denoted by the subtype_mark or access_definition in the declaration of the formal. {*static (subtype)* [partial]} For a formal object of mode **in out**, its type is determined by the subtype_mark or access_definition in the declaration; its nominal subtype is nonstatic, even if the subtype_mark denotes a static subtype; for a composite type, its nominal subtype is unconstrained if the first subtype of the type is unconstrained[, even if the subtype_mark denotes a constrained subtype].

9.a/2 **Reason:** {*AI95-00255-01*} We require that the subtype is unconstrained because a formal **in out** acts like a renaming, and thus the given subtype is ignored for purposes of matching; any value of the type can be passed. Thus we can assume only that the object is constrained if the first subtype is constrained (and thus there can be no unconstrained subtypes for the type). If we didn't do this, it would be possible to rename or take 'Access of components that could disappear due to an assignment to the whole object.

9.b/2 **Discussion:** {*AI95-00423-01*} The two "even if" clauses are OK even though they don't mention access_definitions; an access subtype can neither be a static subtype nor be a composite type.

10/2 {*AI95-00269-01*} {*full constant declaration (corresponding to a formal object of mode in)*} {*stand-alone constant (corresponding to a formal object of mode in)*} {*stand-alone object* [partial]} In an instance, a formal_object_declaration of mode **in** is a *full constant declaration* and declares a new stand-alone constant object whose initialization expression is the actual, whereas a formal_object_declaration of mode **in out** declares a view whose properties are identical to those of the actual.

10.a/2 **Ramification:** {*AI95-00287-01*} These rules imply that generic formal objects of mode **in** are passed by copy (or are built-in-place for a limited type), whereas generic formal objects of mode **in out** are passed by reference.

10.b Initialization and finalization happen for the constant declared by a formal_object_declaration of mode **in** as for any constant; see 3.3.1, "Object Declarations" and 7.6, "User-Defined Assignment and Finalization".

10.c {*subtype (of a generic formal object)* [partial]} In an instance, the subtype of a generic formal object of mode **in** is as for the equivalent constant. In an instance, the subtype of a generic formal object of mode **in out** is the subtype of the corresponding generic actual.

Dynamic Semantics

11 {*evaluation (generic_association for a formal object of mode in)* [partial]} {*assignment operation (during evaluation of a generic_association for a formal object of mode in)*} For the evaluation of a generic_association for a formal object of mode **in**, a constant object is created, the value of the actual parameter is converted to the nominal subtype of the formal object, and assigned to the object[, including any value adjustment — see 7.6]. {*implicit subtype conversion (generic formal object of mode in)* [partial]}

11.a **Ramification:** This includes evaluating the actual and doing a subtype conversion, which might raise an exception.

11.b **Discussion:** The rule for evaluating a generic_association for a formal object of mode **in out** is covered by the general Dynamic Semantics rule in 12.3.

NOTES

12 6 The constraints that apply to a generic formal object of mode **in out** are those of the corresponding generic actual parameter (not those implied by the subtype_mark that appears in the formal_object_declaration). Therefore, to avoid confusion, it is recommended that the name of a first subtype be used for the declaration of such a formal object.

12.a **Ramification:** Constraint checks are done at instantiation time for formal objects of mode **in**, but not for formal objects of mode **in out**.

Extensions to Ada 83

12.b {*extensions to Ada 83*} In Ada 83, it is forbidden to pass a (nongeneric) formal parameter of mode **out**, or a subcomponent thereof, to a generic formal object of mode **in out**. This restriction is removed in Ada 95.

Wording Changes from Ada 83

12.c We make "mode" explicit in the syntax. RM83 refers to the mode without saying what it is. This is also more uniform with the way (nongeneric) formal parameters are defined.

12.d We considered allowing mode **out** in Ada 95, for uniformity with (nongeneric) formal parameters. The semantics would be identical for modes **in out** and **out**. (Note that generic formal objects of mode **in out** are passed by reference. Note that for (nongeneric) formal parameters that are allowed to be passed by reference, the semantics of **in out** and **out** is the same. The difference might serve as documentation. The same would be true for generic formal objects, if **out** were allowed, so it would be consistent.) We decided not to make this change, because it does not produce any important benefit, and any change has some cost.

Extensions to Ada 95

{*AI95-00287-01*} {*extensions to Ada 95*} A generic formal **in** object can have a limited type. The actual for such an object must be built-in-place via a function_call or aggregate, see 7.5. 12.e/2

{*AI95-00423-01*} A generic formal object can have a null_exclusion or an anonymous access type. 12.f/2

Wording Changes from Ada 95

{*AI95-00255-01*} Clarified that the nominal subtype of a composite formal **in out** object is unconstrained if the first subtype of the type is unconstrained. 12.g/2

{*AI95-00269-01*} Clarified that a formal **in** object can be static when referenced from outside of the instance (by declaring such an object to be a full constant declaration). 12.h/2

12.5 Formal Types

{*AI95-00442-01*} [A generic formal subtype can be used to pass to a generic unit a subtype whose type is in a certain category of types.] 1/2

Reason: We considered having intermediate syntactic categories formal_integer_type_definition, formal_real_type_definition, and formal_fixed_point_definition, to be more uniform with the syntax rules for non-generic-formal types. However, that would make the rules for formal types slightly more complicated, and it would cause confusion, since formal_discrete_type_definition would not fit into the scheme very well. 1.a

Syntax

```
formal_type_declaration ::=                                                      2
  type defining_identifier[discriminant_part] is formal_type_definition;

{AI95-00251-01} formal_type_definition ::=                                       3/2
    formal_private_type_definition
  | formal_derived_type_definition
  | formal_discrete_type_definition
  | formal_signed_integer_type_definition
  | formal_modular_type_definition
  | formal_floating_point_definition
  | formal_ordinary_fixed_point_definition
  | formal_decimal_fixed_point_definition
  | formal_array_type_definition
  | formal_access_type_definition
  | formal_interface_type_definition
```

Legality Rules

{*generic actual subtype*} {*actual subtype*} {*generic actual type*} {*actual type*} For a generic formal subtype, the actual shall be a subtype_mark; it denotes the *(generic) actual subtype*. 4

Ramification: When we say simply "formal" or "actual" (for a generic formal that denotes a subtype) we're talking about the subtype, not the type, since a name that denotes a formal_type_declaration denotes a subtype, and the corresponding actual also denotes a subtype. 4.a

Static Semantics

{*generic formal type*} {*formal type*} {*generic formal subtype*} {*formal subtype*} A formal_type_declaration declares a *(generic) formal type*, and its first subtype, the *(generic) formal subtype*. 5

Ramification: A subtype (other than the first subtype) of a generic formal type is not a generic formal subtype. 5.a

{*AI95-00442-01*} {*determined category for a formal type*} {*category determined for a formal type*} The form of a formal_type_definition *determines a category (of types)* to which the formal type belongs. For a formal_private_type_definition the reserved words **tagged** and **limited** indicate the category of types (see 12.5.1). For a formal_derived_type_definition the category of types is the derivation class rooted at the 6/2 ancestor type. For other formal types, the name of the syntactic category indicates the category of types; a formal_discrete_type_definition defines a discrete type, and so on.

6.a **Reason:** This rule is clearer with the flat syntax rule for formal_type_definition given above. Adding formal_integer_type_definition and others would make this rule harder to state clearly.

6.b/2 {*AI95-00442-01*} We use "category' rather than "class" above, because the requirement that classes are closed under derivation is not important here. Moreover, there are interesting categories that are not closed under derivation. For instance, limited and interface are categories that do not form classes.

Legality Rules

7/2 {*AI95-00442-01*} The actual type shall be in the category determined for the formal.

7.a/2 **Ramification:** {*AI95-00442-01*} For example, if the category determined for the formal is the category of all discrete types, then the actual has to be discrete.

7.b/2 {*AI95-00442-01*} Note that this rule does not require the actual to belong to every category to which the formal belongs. For example, formal private types are in the category of composite types, but the actual need not be composite. Furthermore, one can imagine an infinite number of categories that are just arbitrary sets of types (even though we don't give them names, since they are uninteresting). We don't want this rule to apply to *those* categories.

7.c/2 {*AI95-00114-01*} {*AI95-00442-01*} "Limited" is not an "interesting" category, but "nonlimited" is; it is legal to pass a nonlimited type to a limited formal type, but not the other way around. The reserved word **limited** really represents a category containing both limited and nonlimited types. "Private" is not a category for this purpose; a generic formal private type accepts both private and nonprivate actual types.

7.d/2 {*AI95-00442-01*} It is legal to pass a class-wide subtype as the actual if it is in the right category, so long as the formal has unknown discriminants.

Static Semantics

8/2 {*8652/0037*} {*AI95-00043-01*} {*AI95-00233-01*} {*AI95-00442-01*} [The formal type also belongs to each category that contains the determined category.] The primitive subprograms of the type are as for any type in the determined category. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later immediately within the declarative region in which the type is declared according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. [The rules specific to formal derived types are given in 12.5.1.]

8.a/2 **Ramification:** {*AI95-00442-01*} All properties of the type are as for any type in the category. Some examples: The primitive operations available are as defined by the language for each category. The form of constraint applicable to a formal type in a subtype_indication depends on the category of the type as for a nonformal type. The formal type is tagged if and only if it is declared as a tagged private type, or as a type derived from a (visibly) tagged type. (Note that the actual type might be tagged even if the formal type is not.)

NOTES

9 7 Generic formal types, like all types, are not named. Instead, a name can denote a generic formal subtype. Within a generic unit, a generic formal type is considered as being distinct from all other (formal or nonformal) types.

9.a **Proof:** This follows from the fact that each formal_type_declaration declares a type.

10 8 A discriminant_part is allowed only for certain kinds of types, and therefore only for certain kinds of generic formal types. See 3.7.

10.a **Ramification:** The term "formal floating point type" refers to a type defined by a formal_floating_point_definition. It does not include a formal derived type whose ancestor is floating point. Similar terminology applies to the other kinds of formal_type_definition.

Examples

11 *Examples of generic formal types:*

12
```
type Item is private;
type Buffer(Length : Natural) is limited private;
```

13
```
type Enum  is (<>);
type Int   is range <>;
type Angle is delta <>;
type Mass  is digits <>;
```

14
```
type Table is array (Enum) of Item;
```

15 *Example of a generic formal part declaring a formal integer type:*

16
```
generic
   type Rank is range <>;
   First  : Rank := Rank'First;
   Second : Rank := First + 1;  -- the operator "+" of the type Rank
```

Wording Changes from Ada 83

16.a RM83 has separate sections “Generic Formal Xs” and “Matching Rules for Formal Xs” (for various X's) with most of the text redundant between the two. We have combined the two in order to reduce the redundancy. In RM83, there is no “Matching Rules for Formal Types” section; nor is there a “Generic Formal Y Types” section (for Y = Private, Scalar, Array, and Access). This causes, for example, the duplication across all the “Matching Rules for Y Types” sections of the rule that the actual passed to a formal type shall be a subtype; the new organization avoids that problem.

16.b The matching rules are stated more concisely.

16.c We no longer consider the multiplying operators that deliver a result of type *universal_fixed* to be predefined for the various types; there is only one of each in package Standard. Therefore, we need not mention them here as RM83 had to.

Wording Changes from Ada 95

16.d/2 {*8652/0037*} {*AI95-00043-01*} {*AI95-00233-01*} Corrigendum 1 corrected the wording to properly define the location where operators are defined for formal array types. The wording here was inconsistent with that in 7.3.1, “Private Operations”. For the Amendment, this wording was corrected again, because it didn't reflect the Corrigendum 1 revisions in 7.3.1.

16.e/2 {*AI95-00251-01*} Formal interface types are defined; see 12.5.5, “Formal Interface Types”.

16.f/2 {*AI95-00442-01*} We use “determines a category” rather than class, since not all interesting properties form a class.

12.5.1 Formal Private and Derived Types

1/2 {*AI95-00442-01*} [In its most general form, the category determined for a formal private type is all types, but it can be restricted to only nonlimited types or to only tagged types. The category determined for a formal derived type is the derivation class rooted at the ancestor type.]

1.a/2 **Proof:** {*AI95-00442-01*} The first rule is given normatively below, and the second rule is given normatively in 12.5; they are repeated here to give a capsule summary of what this subclause is about.

Syntax

2 formal_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] **private**

3/2 {*AI95-00251-01*} {*AI95-00419-01*} {*AI95-00443-01*} formal_derived_type_definition ::=
[**abstract**] [**limited** | **synchronized**] **new** subtype_mark [[**and** interface_list]**with private**]

Legality Rules

4 If a generic formal type declaration has a known_discriminant_part, then it shall not include a default_expression for a discriminant.

4.a **Ramification:** Consequently, a generic formal subtype with a known_discriminant_part is an indefinite subtype, so the declaration of a stand-alone variable has to provide a constraint on such a subtype, either explicitly, or by its initial value.

5/2 {*AI95-00401-01*} {*AI95-00419-01*} {*AI95-00443-01*} {*ancestor subtype (of a formal derived type)*} {*private extension* [partial]} The *ancestor subtype* of a formal derived type is the subtype denoted by the subtype_mark of the formal_derived_type_definition. For a formal derived type declaration, the reserved words **with private** shall appear if and only if the ancestor type is a tagged type; in this case the formal derived type is a private extension of the ancestor type and the ancestor shall not be a class-wide type. [Similarly, an interface_list or the optional reserved words **abstract** or **synchronized** shall appear only if the ancestor type is a tagged type]. The reserved word **limited** or **synchronized** shall appear only if the ancestor type [and any progenitor types] are limited types. The reserved word **synchronized** shall appear (rather than **limited**) if the ancestor type or any of the progenitor types are synchronized interfaces.

5.a **Reason:** We use the term "ancestor" here instead of "parent" because the actual can be any descendant of the ancestor, not necessarily a direct descendant.

5.b/2 {*AI95-00419-01*} We require the ancestor type to be limited when **limited** appears so that we avoid oddies like limited integer types. Normally, **limited** means "match anything" for a generic formal, but it was felt that allowing limited elementary types to be declared was just too weird. Integer still matches a formal limited private type; it is only a problem when the type is known to be elementary. Note that the progenitors are required to be limited by rules in 3.9.4, thus that part of the rule is redundant.

5.c/2 {*AI95-00443-01*} We require that **synchronized** appear if the ancestor or any of the progenitors are synchronized, so that property is explicitly given in the program text – it is not automatically inherited from the ancestors. However, it can be given even if neither the ancestor nor the progenitors are synchronized.

5.1/2 {*AI95-00251-01*} {*AI95-00401-01*} {*AI95-00443-01*} The actual type for a formal derived type shall be a descendant of [the ancestor type and] every progenitor of the formal type. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

5.d/2 **Proof:** The actual type has to be a descendant of the ancestor type, in order that it be in the correct class. Thus, that part of the rule is redundant.

5.e/2 **Discussion:** For a non-formal private extension, we require the partial view to be synchronized if the full view is synchronized tagged. This does not apply to a formal private extension — it is OK if the formal is not synchronized. Any attempt to extend the formal type will be rechecked in the instance, where the rule disallowing extending a sychronized non-interface type will be enforced. This is consistent with the "no hidden interfaces" rule also applying only to non-formal private extensions, as well as the rule that a limited non-formal private extension implies a limited full type. Formal private extensions are exempted from all these rules to enable the construction of generics that can be used with the widest possible range of types. In particular, an indefinite tagged limited formal private type can match any "concrete" actual tagged type.

6 If the formal subtype is definite, then the actual subtype shall also be definite.

6.a **Ramification:** On the other hand, for an indefinite formal subtype, the actual can be either definite or indefinite.

7 For a generic formal derived type with no discriminant_part:

8 • If the ancestor subtype is constrained, the actual subtype shall be constrained, and shall be statically compatible with the ancestor;

8.a **Ramification:** In other words, any constraint on the ancestor subtype is considered part of the "contract."

9 • If the ancestor subtype is an unconstrained access or composite subtype, the actual subtype shall be unconstrained.

9.a **Reason:** This rule ensures that if a composite constraint is allowed on the formal, one is also allowed on the actual. If the ancestor subtype is an unconstrained scalar subtype, the actual is allowed to be constrained, since a scalar constraint does not cause further constraints to be illegal.

10 • If the ancestor subtype is an unconstrained discriminated subtype, then the actual shall have the same number of discriminants, and each discriminant of the actual shall correspond to a discriminant of the ancestor, in the sense of 3.7.

10.a **Reason:** This ensures that if a discriminant constraint is given on the formal subtype, the corresponding constraint in the instance will make sense, without additional run-time checks. This is not necessary for arrays, since the bounds cannot be overridden in a type extension. An unknown_discriminant_part may be used to relax these matching requirements.

10.1/2 • {*AI95-00231-01*} If the ancestor subtype is an access subtype, the actual subtype shall exclude null if and only if the ancestor subtype excludes null.

Reason: We require that the “excludes null” property match, because it would be difficult to write a correct generic for a formal access type without knowing this property. Many typical algorithms and techniques will not work for a subtype that excludes null (setting an unused component to **null**, default-initialized objects, and so on). We want this sort of requirement to be reflected in the contract of the generic. 10.b/2

The declaration of a formal derived type shall not have a known_discriminant_part. For a generic formal private type with a known_discriminant_part: 11

- The actual type shall be a type with the same number of discriminants. 12
- The actual subtype shall be unconstrained. 13
- The subtype of each discriminant of the actual type shall statically match the subtype of the corresponding discriminant of the formal type. {*statically matching (required)* [partial]} 14

Reason: We considered defining the first and third rule to be called “subtype conformance” for discriminant_parts. We rejected that idea, because it would require implicit (inherited) discriminant_parts, which seemed like too much mechanism. 14.a

[For a generic formal type with an unknown_discriminant_part, the actual may, but need not, have discriminants, and may be definite or indefinite.] 15

Static Semantics

{*AI95-00442-01*} The category determined for a formal private type is as follows: 16/2

Type Definition	*Determined Category*
limited private	the category of all types
private	the category of all nonlimited types
tagged limited private	the category of all tagged types
tagged private	the category of all nonlimited tagged types

17/2

[The presence of the reserved word **abstract** determines whether the actual type may be abstract.] 18

A formal private or derived type is a private or derived type, respectively. A formal derived tagged type is a private extension. [A formal private or derived type is abstract if the reserved word **abstract** appears in its declaration.] 19

{*AI95-00233-01*} If the ancestor type is a composite type that is not an array type, the formal type inherits components from the ancestor type (including discriminants if a new discriminant_part is not specified), as for a derived type defined by a derived_type_definition (see 3.4 and 7.3.1). 20/2

{*8652/0038*} {*AI95-00202*} {*AI95-00233-01*} {*AI95-00401-01*} For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type and any progenitor types, and are implicitly declared at the earliest place, if any, immediately within the declarative region in which the formal type is declared, where the corresponding primitive subprogram of the ancestor or progenitor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the corresponding primitive subprogram of the ancestor or progenitor of the formal derived type, even if this primitive has been overridden for the actual type. When the ancestor or progenitor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor or progenitor. [In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.] 21/2

Ramification: {*AI95-00401-01*} The above rule defining the properties of primitive subprograms in an instance applies even if the subprogram has been overridden or hidden for the actual type. This rule is necessary for untagged types, because their primitive subprograms might have been overridden by operations that are not subtype-conformant with the operations defined for the class. For tagged types, the rule still applies, but the primitive subprograms will dispatch to the appropriate implementation based on the type and tag of the operands. Even for tagged types, the 21.a/2 formal parameter names and default_expressions are determined by those of the primitive subprograms of the specified ancestor type (or progenitor type, for subprograms inherited from an interface type).

22/1 For a prefix S that denotes a formal indefinite subtype, the following attribute is defined:

23 S'Definite S'Definite yields True if the actual subtype corresponding to S is definite; otherwise it yields False. The value of this attribute is of the predefined type Boolean.

23.a/2 **Discussion:** {*AI95-00114-01*} Whether an actual subtype is definite or indefinite may have a major effect on the algorithm used in a generic. For example, in a generic I/O package, whether to use fixed-length or variable-length records could depend on whether the actual is definite or indefinite. This attribute is essentially a replacement for the Constrained attribute, which is now considered obsolete.

Dynamic Semantics

23.1/2 {*AI95-00158-01*} In the case where a formal type is tagged with unknown discriminants, and the actual type is a class-wide type *T*'Class:

23.2/2
- {*AI95-00158-01*} For the purposes of defining the primitive operations of the formal type, each of the primitive operations of the actual type is considered to be a subprogram (with an intrinsic calling convention — see 6.3.1) whose body consists of a dispatching call upon the corresponding operation of *T*, with its formal parameters as the actual parameters. If it is a function, the result of the dispatching call is returned.

23.3/2
- {*AI95-00158-01*} If the corresponding operation of *T* has no controlling formal parameters, then the controlling tag value is determined by the context of the call, according to the rules for tag-indeterminate calls (see 3.9.2 and 5.2). In the case where the tag would be statically determined to be that of the formal type, the call raises Program_Error. If such a function is renamed, any call on the renaming raises Program_Error. {*Program_Error (raised by failure of run-time check)*}

23.b/2 **Discussion:** As it states in 6.3.1, the convention of an inherited subprogram of a generic formal tagged type with unknown discriminants is intrinsic.

23.c/2 In the case of a corresponding primitive of T with no controlling formal parameters, the context of the call provides the controlling tag value for the dispatch. If no tag is provided by context, Program_Error is raised rather than resorting to a nondispatching call. For example:

23.d/2
```
generic
   type NT(<>) is new T with private;
    -- Assume T has operation "function Empty return T;"
package G is
   procedure Test(X : in out NT);
end G;
```

23.e/2
```
package body G is
   procedure Test(X : in out NT) is
   begin
      X := Empty;   -- Dispatching based on X'Tag takes
                    -- place if actual is class-wide.
      declare
          Y : NT := Empty;
                    -- If actual is class-wide, this raises Program_Error
                    -- as there is no tag provided by context.
      begin
          X := Y;   -- We never get this far.
      end;
   end Test;
end G;
```

23.f/2
```
type T1 is new T with null record;
package I is new G(T1'Class);
```

NOTES

24/2 9 {*AI95-00442-01*} In accordance with the general rule that the actual type shall belong to the category determined for the formal (see 12.5, "Formal Types"):

25
- If the formal type is nonlimited, then so shall be the actual;

26
- For a formal derived type, the actual shall be in the class rooted at the ancestor subtype.

27 10 The actual type can be abstract only if the formal type is abstract (see 3.9.3).

Reason: This is necessary to avoid contract model problems, since one or more of its primitive subprograms are abstract; it is forbidden to create objects of the type, or to declare functions returning the type. 27.a

Ramification: On the other hand, it is OK to pass a non-abstract actual to an abstract formal — **abstract** on the formal indicates that the actual might be abstract. 27.b

11 If the formal has a discriminant_part, the actual can be either definite or indefinite. Otherwise, the actual has to be definite. 28

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} Ada 83 does not have unknown_discriminant_parts, so it allows indefinite subtypes to be passed to definite formals, and applies a legality rule to the instance body. This is a contract model violation. Ada 95 disallows such cases at the point of the instantiation. The workaround is to add (<>) as the discriminant_part of any formal subtype if it is intended to be used with indefinite actuals. If that's the intent, then there can't be anything in the generic body that would require a definite subtype. 28.a

The check for discriminant subtype matching is changed from a run-time check to a compile-time check. 28.b

Extensions to Ada 95

{*AI95-00251-01*} {*AI95-00401-01*} {*AI95-00419-01*} {*AI95-00443-01*} {*extensions to Ada 95*} A generic formal derived type can include progenitors (interfaces) as well as a primary ancestor. It also may include **limited** to indicate that it is a limited type, and **synchronized** to indicate that it is a synchronized type. 28.c/2

Wording Changes from Ada 95

{*8652/0038*} {*AI95-00202-01*} **Corrigendum:** Corrected wording to define the operations that are inherited when the ancestor of a formal type is itself a formal type to avoid anomalies. 28.d/2

{*AI95-00158-01*} Added a semantic description of the meaning of operations of an actual class-wide type, as such a type does not have primitive operations of its own. 28.e/2

{*AI95-00231-01*} Added a matching rule for access subtypes that exclude null. 28.f/2

{*AI95-00233-01*} The wording for the declaration of implicit operations is corrected to be consistent with 7.3.1 as modified by Corrigendum 1. 28.g/2

{*AI95-00442-01*} We change to "determines a category" as that is the new terminology (it avoids confusion, since not all interesting properties form a class). 28.h/2

12.5.2 Formal Scalar Types

{*AI95-00442-01*} A *formal scalar type* is one defined by any of the formal_type_definitions in this subclause. [The category determined for a formal scalar type is the category of all discrete, signed integer, modular, floating point, ordinary fixed point, or decimal types.] 1/2

Proof: {*AI95-00442-01*} The second rule follows from the rule in 12.5 that says that the category is determined by the one given in the name of the syntax production. The effect of the rule is repeated here to give a capsule summary of what this subclause is about. 1.a/2

Ramification: {*AI95-00442-01*} The "category of a type" includes any classes that the type belongs to. 1.b/2

Syntax

formal_discrete_type_definition ::= (<>) 2

formal_signed_integer_type_definition ::= **range** <> 3

formal_modular_type_definition ::= **mod** <> 4

formal_floating_point_definition ::= **digits** <> 5

formal_ordinary_fixed_point_definition ::= **delta** <> 6

formal_decimal_fixed_point_definition ::= **delta** <> **digits** <> 7

Legality Rules

The actual type for a formal scalar type shall not be a nonstandard numeric type. 8

8.a **Reason:** This restriction is necessary because nonstandard numeric types have some number of restrictions on their use, which could cause contract model problems in a generic body. Note that nonstandard numeric types can be passed to formal derived and formal private subtypes, assuming they obey all the other rules, and assuming the implementation allows it (being nonstandard means the implementation might disallow anything).

NOTES

9 12 The actual type shall be in the class of types implied by the syntactic category of the formal type definition (see 12.5, “Formal Types”). For example, the actual for a formal_modular_type_definition shall be a modular type.

Wording Changes from Ada 95

9.a/2 {*AI95-00442-01*} We change to “determines a category” as that is the new terminology (it avoids confusion, since not all interesting properties form a class).

12.5.3 Formal Array Types

1/2 {*AI95-00442-01*} [The category determined for a formal array type is the category of all array types.]

1.a/2 **Proof:** {*AI95-00442-01*} This rule follows from the rule in 12.5 that says that the category is determined by the one given in the name of the syntax production. The effect of the rule is repeated here to give a capsule summary of what this subclause is about.

Syntax

2 formal_array_type_definition ::= array_type_definition

Legality Rules

3 The only form of discrete_subtype_definition that is allowed within the declaration of a generic formal (constrained) array subtype is a subtype_mark.

3.a **Reason:** The reason is the same as for forbidding constraints in subtype_indications (see 12.1).

4 For a formal array subtype, the actual subtype shall satisfy the following conditions:

5 • The formal array type and the actual array type shall have the same dimensionality; the formal subtype and the actual subtype shall be either both constrained or both unconstrained.

6 • For each index position, the index types shall be the same, and the index subtypes (if unconstrained), or the index ranges (if constrained), shall statically match (see 4.9.1). {*statically matching (required)* [partial]}

7 • The component subtypes of the formal and actual array types shall statically match. {*statically matching (required)* [partial]}

8 • If the formal type has aliased components, then so shall the actual.

8.a **Ramification:** On the other hand, if the formal's components are not aliased, then the actual's components can be either aliased or not.

Examples

9 *Example of formal array types:*

10
```
--  given the generic package
```

11
```
generic
   type Item   is private;
   type Index  is (<>);
   type Vector is array (Index range <>) of Item;
   type Table  is array (Index) of Item;
package P is
   ...
end P;
```

12
```
--  and the types
```

13
```
type Mix    is array (Color range <>) of Boolean;
type Option is array (Color) of Boolean;
```

```
-- then Mix can match Vector and Option can match Table                14
```

```
package R is new P(Item   => Boolean, Index => Color,                  15
                   Vector => Mix,     Table => Option);
```

```
-- Note that Mix cannot match Table and Option cannot match Vector     16
```

Incompatibilities With Ada 83

16.a {*incompatibilities with Ada 83*} The check for matching of component subtypes and index subtypes or index ranges is changed from a run-time check to a compile-time check. The Ada 83 rule that "If the component type is not a scalar type, then the component subtypes shall be either both constrained or both unconstrained" is removed, since it is subsumed by static matching. Likewise, the rules requiring that component types be the same is subsumed.

Wording Changes from Ada 95

16.b/2 {*AI95-00442-01*} We change to "determines a category" as that is the new terminology (it avoids confusion, since not all interesting properties form a class).

12.5.4 Formal Access Types

1/2 {*AI95-00442-01*} [The category determined for a formal access type is the category of all access types.]

1.a/2 **Proof:** {*AI95-00442-01*} This rule follows from the rule in 12.5 that says that the category is determined by the one given in the name of the syntax production. The effect of the rule is repeated here to give a capsule summary of what this subclause is about.

Syntax

2 formal_access_type_definition ::= access_type_definition

Legality Rules

3 For a formal access-to-object type, the designated subtypes of the formal and actual types shall statically match. {*statically matching (required)* [partial]}

4/2 {*AI95-00231-01*} If and only if the general_access_modifier **constant** applies to the formal, the actual shall be an access-to-constant type. If the general_access_modifier **all** applies to the formal, then the actual shall be a general access-to-variable type (see 3.10). If and only if the formal subtype excludes null, the actual subtype shall exclude null.

4.a **Ramification:** If no _modifier applies to the formal, then the actual type may be either a pool-specific or a general access-to-variable type.

4.a.1/1 **Reason:** {*8652/0109*} {*AI95-00025-01*} Matching an access-to-variable to a formal access-to-constant type cannot be allowed. If it were allowed, it would be possible to create an access-to-variable value designating a constant.

4.b/2 {*AI95-00231-01*} We require that the "excludes null" property match, because it would be difficult to write a correct generic for a formal access type without knowing this property. Many typical algorithms and techniques will not work for a subtype that excludes null (setting an unused component to **null**, default-initialized objects, and so on). Even Ada.Unchecked_Deallocation would fail for a subtype that excludes null. Most generics would end up with comments saying that they are not intended to work for subtypes that exclude null. We would rather that this sort of requirement be reflected in the contract of the generic.

5 For a formal access-to-subprogram subtype, the designated profiles of the formal and the actual shall be mode-conformant, and the calling convention of the actual shall be *protected* if and only if that of the formal is *protected*. {*mode conformance (required)*}

5.a **Reason:** We considered requiring subtype conformance here, but mode conformance is more flexible, given that there is no way in general to specify the convention of the formal.

Examples

6 *Example of formal access types:*

```
-- the formal types of the generic package                             7
```

8
```
generic
   type Node is private;
   type Link is access Node;
package P is
   ...
end P;
```

9 -- *can be matched by the actual types*

10
```
type Car;
type Car_Name is access Car;
```

11
```
type Car is
   record
      Pred, Succ : Car_Name;
      Number     : License_Number;
      Owner      : Person;
   end record;
```

12 -- *in the following generic instantiation*

13
```
package R is new P(Node => Car, Link => Car_Name);
```

Incompatibilities With Ada 83

13.a {*incompatibilities with Ada 83*} The check for matching of designated subtypes is changed from a run-time check to a compile-time check. The Ada 83 rule that “If the designated type is other than a scalar type, then the designated subtypes shall be either both constrained or both unconstrained” is removed, since it is subsumed by static matching.

Extensions to Ada 83

13.b {*extensions to Ada 83*} Formal access-to-subprogram subtypes and formal general access types are new concepts.

Wording Changes from Ada 95

13.c/2 {*AI95-00231-01*} Added a matching rule for subtypes that exclude null.

13.d/2 {*AI95-00442-01*} We change to “determines a category” as that is the new terminology (it avoids confusion, since not all interesting properties form a class).

12.5.5 Formal Interface Types

1/2 {*AI95-00251-01*} {*AI95-00442-01*} [The category determined for a formal interface type is the category of all interface types.]

1.a/2 **Proof:** {*AI95-00442-01*} This rule follows from the rule in 12.5 that says that the category is determined by the one given in the name of the syntax production. The effect of the rule is repeated here to give a capsule summary of what this subclause is about.

1.b/2 **Ramification:** Here we're taking advantage of our switch in terminology from “determined class” to “determined category”; by saying “category” rather than “class”, we require that any actual type be an interface type, not just some type derived from an interface type.

Syntax

2/2 {*AI95-00251-01*} formal_interface_type_definition ::= interface_type_definition

Legality Rules

3/2 {*AI95-00251*} {*AI95-00401*} The actual type shall be a descendant of every progenitor of the formal type.

4/2 {*AI95-00345*} The actual type shall be a limited, task, protected, or synchronized interface if and only if the formal type is also, respectively, a limited, task, protected, or synchronized interface.

4.a/2 **Discussion:** We require the kind of interface type to match exactly because without that it is almost impossible to properly implement the interface.

Examples

```
{AI95-00433-01} type Root_Work_Item is tagged private;                5/2
```

```
{AI95-00433-01} generic                                               6/2
   type Managed_Task is task interface;
   type Work_Item(<>) is new Root_Work_Item with private;
package Server_Manager is
   task type Server is new Managed_Task with
      entry Start(Data : in out Work_Item);
   end Server;
end Server_Manager;
```

{*AI95-00433-01*} This generic allows an application to establish a standard interface that all tasks need to implement so they can be managed appropriately by an application-specific scheduler. 7/2

Extensions to Ada 95

{*AI95-00251-01*} {*AI95-00345-01*} {*AI95-00401-01*} {*AI95-00442-01*} {*extensions to Ada 95*} The formal interface type is new. 7.a/2

12.6 Formal Subprograms

[{*generic formal subprogram*} {*formal subprogram, generic*} Formal subprograms can be used to pass callable entities to a generic unit.] 1

Language Design Principles

Generic formal subprograms are like renames of the explicit_generic_actual_parameter. 1.a

Syntax

{*AI95-00260-02*} formal_subprogram_declaration ::= formal_concrete_subprogram_declaration 2/2
   | formal_abstract_subprogram_declaration

{*AI95-00260-02*} formal_concrete_subprogram_declaration ::= 2.1/2
   with subprogram_specification [**is** subprogram_default];

{*AI95-00260-02*} formal_abstract_subprogram_declaration ::= 2.2/2
   with subprogram_specification **is abstract** [subprogram_default];

{*AI95-00348-01*} subprogram_default ::= default_name | <> | **null** 3/2

default_name ::= name 4

{*AI95-00260-02*} {*AI95-00348-01*} A subprogram_default of **null** shall not be specified for a formal function or for a formal_abstract_subprogram_declaration. 4.1/2

Reason: There are no null functions because the return value has to be constructed somehow. We don't allow null for abstract formal procedures, as the operation is dispatching. It doesn't seem appropriate (or useful) to say that the implementation of something is null in the formal type and all possible descendants of that type. This also would define a dispatching operation that doesn't correspond to a slot in the tag of the controlling type, which would be a new concept. Finally, additional rules would be needed to define the meaning of a dispatching null procedure (for instance, the convention of such a subprogram should be intrinsic, but that's not what the language says). It doesn't seem worth the effort. 4.a/2

Name Resolution Rules

{*expected profile (formal subprogram default_name)* [partial]} The expected profile for the default_name, if any, is that of the formal subprogram. 5

Ramification: This rule, unlike others in this clause, is observed at compile time of the generic_declaration. 5.a

The evaluation of the default_name takes place during the elaboration of each instantiation that uses the default, as defined in 12.3, “Generic Instantiation”. 5.b

6 {*expected profile (formal subprogram actual)* [partial]} For a generic formal subprogram, the expected profile for the actual is that of the formal subprogram.

Legality Rules

7 The profiles of the formal and any named default shall be mode-conformant. {*mode conformance (required)*}

7.a **Ramification:** This rule, unlike others in this clause, is checked at compile time of the generic_declaration.

8 The profiles of the formal and actual shall be mode-conformant. {*mode conformance (required)*}

8.1/2 {*AI95-00423-01*} For a parameter or result subtype of a formal_subprogram_declaration that has an explicit null_exclusion:

8.2/2
- if the actual matching the formal_subprogram_declaration denotes a generic formal object of another generic unit *G*, and the instantiation containing the actual that occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of the generic unit *G*, then the corresponding parameter or result type of the formal subprogram of *G* shall have a null_exclusion;

8.3/2
- otherwise, the subtype of the corresponding parameter or result type of the actual matching the formal_subprogram_declaration shall exclude null. {*generic contract issue* [partial]} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

8.a/2 **Reason:** This rule prevents "lying". **Null** must never be the value of a parameter or result with an explicit null_exclusion. The first bullet is an assume-the-worst rule which prevents trouble in generic bodies (including bodies of child generics) when the formal subtype excludes null implicitly.

8.4/2 {*AI95-00260-02*} If a formal parameter of a formal_abstract_subprogram_declaration is of a specific tagged type *T* or of an anonymous access type designating a specific tagged type *T*, *T* is called a *controlling type* of the formal_abstract_subprogram_declaration. Similarly, if the result of a formal_-abstract_subprogram_declaration for a function is of a specific tagged type *T* or of an anonymous access type designating a specific tagged type *T*, *T* is called a controlling type of the formal_abstract_-subprogram_declaration. A formal_abstract_subprogram_declaration shall have exactly one controlling type. {*controlling type (of a formal_abstract_subprogram_declaration)*}

8.b/2 **Ramification:** The specific tagged type could be any of a formal tagged private type, a formal derived type, a formal interface type, or a normal tagged type. While the last case doesn't seem to be very useful, there isn't any good reason for disallowing it. This rule ensures that the operation is a dispatching operation of some type, and that we unambiguously know what that type is.

8.c/2 We informally call a subprogram declared by a formal_abstract_subprogram_declaration an *abstract formal subprogram*, but we do not use this term in normative wording. {*abstract formal subprogram*} (We do use it often in these notes.)

8.5/2 {*AI95-00260-02*} The actual subprogram for a formal_abstract_subprogram_declaration shall be a dispatching operation of the controlling type or of the actual type corresponding to the controlling type.

8.d/2 **To be honest:** We mean the controlling type of the formal_abstract_subprogram_declaration, of course. Saying that gets unwieldy and redundant (so says at least one reviewer, anyway).

8.e/2 **Ramification:** This means that the actual is either a primitive operation of the controlling type, or an abstract formal subprogram. Also note that this prevents the controlling type from being class-wide (with one exception explained below), as only specific types have primitive operations (and a formal subprogram eventually has to have an actual that is a primitive of some type). This could happen in a case like:

8.f/2
```
generic
   type T(<>) is tagged private;
   with procedure Foo (Obj : in T) is abstract;
package P ...
```

8.g/2
```
package New_P is new P (Something'Class, Some_Proc);
```

The instantiation here is always illegal, because Some_Proc could never be a primitive operation of Something'Class (there are no such operations). That's good, because we want calls to Foo always to be dispatching calls. 8.h/2

Since it is possible for a formal tagged type to be instantiated with a class-wide type, it is possible for the (real) controlling type to be class-wide in one unusual case: 8.i/2

8.j/2
```
generic
   type NT(<>) is new T with private;
   -- Presume that T has the following primitive operation:
   -- with procedure Bar (Obj : in T);
package Gr ...
```

8.k/2
```
package body Gr is
   package New_P2 is new P (NT, Foo => Bar);
end Gr;
```

8.l/2
```
package New_Gr is new Gr (Something'Class);
```

The instantiation of New_P2 is legal, since Bar is a dispatching operation of the actual type of the controlling type of the abstract formal subprogram Foo. This is not a problem, since the rules given in 12.5.1 explain how this routine dispatches even though its parameter is class-wide. 8.m/2

Note that this legality rule never needs to be rechecked in an instance (that contains a nested instantiation). The rule only talks about the actual type of the instantiation; it does not require looking further; if the actual type is in fact a formal type, we do not intend looking at the actual for that formal. 8.n/2

Static Semantics

A formal_subprogram_declaration declares a generic formal subprogram. The types of the formal parameters and result, if any, of the formal subprogram are those determined by the subtype_marks given in the formal_subprogram_declaration; however, independent of the particular subtypes that are denoted by the subtype_marks, the nominal subtypes of the formal parameters and result, if any, are defined to be nonstatic, and unconstrained if of an array type [(no applicable index constraint is provided in a call on a formal subprogram)]. In an instance, a formal_subprogram_declaration declares a view of the actual. The profile of this view takes its subtypes and calling convention from the original profile of the actual entity, while taking the formal parameter names and default_expressions from the profile given in the formal_subprogram_declaration. The view is a function or procedure, never an entry. 9

Discussion: This rule is intended to be the same as the one for renamings-as-declarations, where the formal_subprogram_declaration is analogous to a renaming-as-declaration, and the actual is analogous to the renamed view. 9.a

If a generic unit has a subprogram_default specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal. 10

{*AI95-00348-01*} If a generic unit has a subprogram_default specified by the reserved word **null**, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a null procedure having the profile given in the formal_subprogram_declaration. 10.1/2

{*AI95-00260-02*} The subprogram declared by a formal_abstract_subprogram_declaration with a controlling type *T* is a dispatching operation of type *T*. 10.2/2

Reason: This is necessary to trigger all of the dispatching operation rules. It otherwise would not be considered a dispatching operation, as formal subprograms are never primitive operations. 10.a.1/2

NOTES

13 The matching rules for formal subprograms state requirements that are similar to those applying to subprogram_renaming_declarations (see 8.5.4). In particular, the name of a parameter of the formal subprogram need not be the same as that of the corresponding parameter of the actual subprogram; similarly, for these parameters, default_expressions need not correspond. 11

14 The constraints that apply to a parameter of a formal subprogram are those of the corresponding formal parameter of the matching actual subprogram (not those implied by the corresponding subtype_mark in the _specification of the formal subprogram). A similar remark applies to the result of a function. Therefore, to avoid confusion, it is recommended that the name of a first subtype be used in any declaration of a formal subprogram. 12

13 15 The subtype specified for a formal parameter of a generic formal subprogram can be any visible subtype, including a generic formal subtype of the same generic_formal_part.

14 16 A formal subprogram is matched by an attribute of a type if the attribute is a function with a matching specification. An enumeration literal of a given type matches a parameterless formal function whose result type is the given type.

15 17 A default_name denotes an entity that is visible or directly visible at the place of the generic_declaration; a box used as a default is equivalent to a name that denotes an entity that is directly visible at the place of the _instantiation.

15.a **Proof:** Visibility and name resolution are applied to the equivalent explicit actual parameter.

16/2 18 {*AI95-00260-02*} The actual subprogram cannot be abstract unless the formal subprogram is a formal_abstract_subprogram_declaration (see 3.9.3).

16.1/2 19 {*AI95-00260-02*} The subprogram declared by a formal_abstract_subprogram_declaration is an abstract subprogram. All calls on a subprogram declared by a formal_abstract_subprogram_declaration must be dispatching calls. See 3.9.3.

16.2/2 20 {*AI95-00348-01*} A null procedure as a subprogram default has convention Intrinsic (see 6.3.1).

16.a.1/2 **Proof:** This is an implicitly declared subprogram, so it has convention Intrinsic as defined in 6.3.1.

Examples

17 *Examples of generic formal subprograms:*

18/2
```
{AI95-00433-01} with function "+"(X, Y : Item) return Item is <>;
with function Image(X : Enum) return String is Enum'Image;
with procedure Update is Default_Update;
with procedure Pre_Action(X : in Item) is null;  -- defaults to no action
with procedure Write(S    : not null access Root_Stream_Type'Class;
                     Desc : Descriptor)
                     is abstract Descriptor'Write;  -- see 13.13.2
-- Dispatching operation on Descriptor with default
```

19
```
--  given the generic procedure declaration
```

20
```
generic
   with procedure Action (X : in Item);
procedure Iterate(Seq : in Item_Sequence);
```

21
```
--  and the procedure
```

22
```
procedure Put_Item(X : in Item);
```

23
```
--  the following instantiation is possible
```

24
```
procedure Put_List is new Iterate(Action => Put_Item);
```

Extensions to Ada 95

24.a/2 {*AI95-00260-02*} {*extensions to Ada 95*} The formal_abstract_subprogram_declaration is new. It allows the passing of dispatching operations to generic units.

24.b/2 {*AI95-00348-01*} The formal subprogram default of **null** is new. It allows the default of a generic procedure to do nothing, such as for passing a debugging routine.

Wording Changes from Ada 95

24.c/2 {*AI95-00423-01*} Added matching rules for null_exclusions.

12.7 Formal Packages

1 [{*generic formal package*} {*formal package, generic*} Formal packages can be used to pass packages to a generic unit. The formal_package_declaration declares that the formal package is an instance of a given generic package. Upon instantiation, the actual package has to be an instance of that generic package.]

Syntax

2 formal_package_declaration ::=
  with package defining_identifier **is new** *generic_package*_name formal_package_actual_part;

{*AI95-00317-01*} formal_package_actual_part ::= 3/2
    ([**others** =>] <>)
  | [generic_actual_part]
  | (formal_package_association {, formal_package_association} [, **others** => <>])

{*AI95-00317-01*} formal_package_association ::= 3.1/2
    generic_association
  | *generic_formal_parameter_*selector_name => <>

{*AI95-00317-01*} Any positional formal_package_associations shall precede any named formal_package_associations. 3.2/2

Legality Rules

{*template (for a formal package)*} The *generic_package_*name shall denote a generic package (the *template* for the formal package); the formal package is an instance of the template. 4

{*AI95-00398-01*} A formal_package_actual_part shall contain at most one formal_package_association for each formal parameter. If the formal_package_actual_part does not include "**others** => <>", each formal parameter without an association shall have a default_expression or subprogram_default. 4.1/2

{*AI95-00317-01*} The actual shall be an instance of the template. If the formal_package_actual_part is (<>) or (**others** => <>), [then the actual may be any instance of the template]; otherwise, certain of the actual parameters of the actual instance shall match the corresponding actual parameters of the formal package, determined as follows: 5/2

- {*AI95-00317-01*} If the formal_package_actual_part includes generic_associations as well as associations with <>, then only the actual parameters specified explicitly with generic_associations are required to match; 5.1/2
- {*AI95-00317-01*} Otherwise, all actual parameters shall match[, whether any actual parameter is given explicitly or by default]. 5.2/2

{*AI95-00317-01*} The rules for matching of actual parameters between the actual instance and the formal package are as follows: 5.3/2

- {*AI95-00317-01*} For a formal object of mode **in**, the actuals match if they are static expressions with the same value, or if they statically denote the same constant, or if they are both the literal **null**. 6/2

    Reason: We can't simply require full conformance between the two actual parameter expressions, because the two expressions are being evaluated at different times. 6.a

- For a formal subtype, the actuals match if they denote statically matching subtypes. {*statically matching (required)* [partial]} 7
- For other kinds of formals, the actuals match if they statically denote the same entity. 8

{*8652/0039*} {*AI95-00213-01*} For the purposes of matching, any actual parameter that is the name of a formal object of mode **in** is replaced by the formal object's actual expression (recursively). 8.1/1

Static Semantics

A formal_package_declaration declares a generic formal package. 9

{*AI95-00317-01*} {*visible part (of a formal package)* [partial]} The visible part of a formal package includes the first list of basic_declarative_items of the package_specification. In addition, for each actual parameter that is not required to match, a copy of the declaration of the corresponding formal parameter of the template is included in the visible part of the formal package. If the copied declaration is for a 10/2 formal type, copies of the implicit declarations of the primitive subprograms of the formal type are also included in the visible part of the formal package.

10.a/2 **Ramification:** {*AI95-00317-01*} If the formal_package_actual_part is (<>), then the declarations that occur immediately within the generic_formal_part of the template for the formal package are visible outside the formal package, and can be denoted by expanded names outside the formal package.If only some of the actual parameters are given by <>, then the declaration corresponding to those parameters (but not the others) are made visible.

10.b **Reason:** We always want either the actuals or the formals of an instance to be namable from outside, but never both. If both were namable, one would get some funny anomalies since they denote the same entity, but, in the case of types at least, they might have different and inconsistent sets of primitive operators due to predefined operator “reemergence.” Formal derived types exacerbate the difference. We want the implicit declarations of the generic_formal_part as well as the explicit declarations, so we get operations on the formal types.

10.c **Ramification:** A generic formal package is a package, and is an instance. Hence, it is possible to pass a generic formal package as an actual to another generic formal package.

11/2 {*AI95-00317-01*} For the purposes of matching, if the actual instance *A* is itself a formal package, then the actual parameters of *A* are those specified explicitly or implicitly in the formal_package_actual_part for *A*, plus, for those not specified, the copies of the formal parameters of the template included in the visible part of *A*.

Examples

12/2 {*AI95-00433-01*} *Example of a generic package with formal package parameters:*

13/2
```
with Ada.Containers.Ordered_Maps;  -- see A.18.6
generic
   with package Mapping_1 is new Ada.Containers.Ordered_Maps(<>);
   with package Mapping_2 is new Ada.Containers.Ordered_Maps
                                    (Key_Type => Mapping_1.Element_Type,
                                     others => <>);
package Ordered_Join is
   -- Provide a "join" between two mappings
```
14/2
```
   subtype Key_Type is Mapping_1.Key_Type;
   subtype Element_Type is Mapping_2.Element_Type;
```
15/2
```
   function Lookup(Key : Key_Type) return Element_Type;
```
16/2
```
   ...
end Ordered_Join;
```

17/2 {*AI95-00433-01*} *Example of an instantiation of a package with formal packages:*

18/2
```
with Ada.Containers.Ordered_Maps;
package Symbol_Package is
```
19/2
```
   type String_Id is ...
```
20/2
```
   type Symbol_Info is ...
```
21/2
```
   package String_Table is new Ada.Containers.Ordered_Maps
           (Key_Type => String,
            Element_Type => String_Id);
```
22/2
```
   package Symbol_Table is new Ada.Containers.Ordered_Maps
           (Key_Type => String_Id,
            Element_Type => Symbol_Info);
```
23/2
```
   package String_Info is new Ordered_Join(Mapping_1 => String_Table,
                                           Mapping_2 => Symbol_Table);
```
24/2
```
   Apple_Info : constant Symbol_Info := String_Info.Lookup("Apple");
```
25/2
```
end Symbol_Package;
```

Extensions to Ada 83

25.a {*extensions to Ada 83*} Formal packages are new to Ada 95.

Extensions to Ada 95

{*AI95-00317-01*} {*AI95-00398-01*} {*extensions to Ada 95*} It's now allowed to mix actuals of a formal package that are specified with those that are not specified. 25.b/2

Wording Changes from Ada 95

{*8652/0039*} {*AI95-00213-01*} **Corrigendum:** Corrected the description of formal package matching to say that formal parameters are always replaced by their actual parameters (recursively). This matches the actual practice of compilers, as the ACATS has always required this behavior. 25.c/2

{*AI95-00317-01*} The description of which operations are visible in a formal package has been clarified. We also specify how matching is done when the actual is a formal package. 25.d/2

12.8 Example of a Generic Package

The following example provides a possible formulation of stacks by means of a generic package. The size of each stack and the type of the stack elements are provided as generic formal parameters. 1

Examples

This paragraph was deleted. 2/1

3
```
generic
   Size : Positive;
   type Item is private;
package Stack is
   procedure Push(E : in  Item);
   procedure Pop (E : out Item);
   Overflow, Underflow : exception;
end Stack;
```

4
```
package body Stack is
```

5
```
   type Table is array (Positive range <>) of Item;
   Space : Table(1 .. Size);
   Index : Natural := 0;
```

6
```
   procedure Push(E : in Item) is
   begin
      if Index >= Size then
         raise Overflow;
      end if;
      Index := Index + 1;
      Space(Index) := E;
   end Push;
```

7
```
   procedure Pop(E : out Item) is
   begin
      if Index = 0 then
         raise Underflow;
      end if;
      E := Space(Index);
      Index := Index - 1;
   end Pop;
```

8
```
end Stack;
```

Instances of this generic package can be obtained as follows: 9

10
```
package Stack_Int  is new Stack(Size => 200, Item => Integer);
package Stack_Bool is new Stack(100, Boolean);
```

Thereafter, the procedures of the instantiated packages can be called as follows: 11

12
```
Stack_Int.Push(N);
Stack_Bool.Push(True);
```

13 Alternatively, a generic formulation of the type Stack can be given as follows (package body omitted):

14
```
generic
   type Item is private;
package On_Stacks is
   type Stack(Size : Positive) is limited private;
   procedure Push(S : in out Stack; E : in  Item);
   procedure Pop (S : in out Stack; E : out Item);
   Overflow, Underflow : exception;
private
   type Table is array (Positive range <>) of Item;
   type Stack(Size : Positive) is
      record
         Space : Table(1 .. Size);
         Index : Natural := 0;
      end record;
end On_Stacks;
```

15 In order to use such a package, an instance has to be created and thereafter stacks of the corresponding type can be declared:

16
```
declare
   package Stack_Real is new On_Stacks(Real); use Stack_Real;
   S : Stack(100);
begin
   ...
   Push(S, 2.54);
   ...
end;
```

Section 13: Representation Issues

{*8652/0009*} {*AI95-00137-01*} [This section describes features for querying and controlling certain aspects of entities and for interfacing to hardware.] 1/1

Wording Changes from Ada 83

The clauses of this section have been reorganized. This was necessary to preserve a logical order, given the new Ada 95 semantics given in this section. 1.a

13.1 Operational and Representation Items

{*8652/0009*} {*AI95-00137-01*} [Representation and operational items can be used to specify aspects of entities. Two kinds of aspects of entities can be specified: aspects of representation and operational aspects. Representation items specify how the types and other entities of the language are to be mapped onto the underlying machine. Operational items specify other properties of entities.] 0.1/1

{*8652/0009*} {*AI95-00137-01*} {*representation item*} {*representation pragma* [distributed]} {*pragma, representation* [distributed]} There are six kinds of *representation items*: attribute_definition_clauses for representation attributes, enumeration_representation_clauses, record_representation_clauses, at_clauses, component_clauses, and *representation pragmas*. [They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware).] 1/1

{*8652/0009*} {*AI95-00137-01*} An {*operational item*} *operational item* is an attribute_definition_clause for an operational attribute. 1.1/1

{*8652/0009*} {*AI95-00137-01*} [An operational item or a representation item applies to an entity identified by a local_name, which denotes an entity declared local to the current declarative region, or a library unit declared immediately preceding a representation pragma in a compilation.] 1.2/1

Language Design Principles

{*8652/0009*} {*AI95-00137-01*} Aspects of representation are intended to refer to properties that need to be known before the compiler can generate code to create or access an entity. For instance, the size of an object needs to be known before the object can be created. Conversely, operational aspects are those that only need to be known before they can be used. For instance, how an object is read from a stream only needs to be known when a stream read is executed. Thus, aspects of representation have stricter rules as to when they can be specified. 1.a.1/1

{*AI95-00291-02*} Confirming the value of an aspect with an operational or representation item should never change the semantics of the aspect. Thus Size = 8 (for example) means the same thing whether it was specified with a representation item or whether the compiler chose this value by default. 1.a.2/2

Syntax

{*8652/0009*} {*AI95-00137-01*} aspect_clause ::= attribute_definition_clause 2/1
| enumeration_representation_clause
| record_representation_clause
| at_clause

local_name ::= direct_name 3
| direct_name'attribute_designator
| *library_unit*_name

{*8652/0009*} {*AI95-00137-01*} A representation pragma is allowed only at places where an aspect_clause or compilation_unit is allowed. {*representation_clause: See aspect_clause*} 4/1

Name Resolution Rules

5/1 {*8652/0009*} {*AI95-00137-01*} In an operational item or representation item, if the local_name is a direct_name, then it shall resolve to denote a declaration (or, in the case of a pragma, one or more declarations) that occurs immediately within the same declarative region as the item. If the local_name has an attribute_designator, then it shall resolve to denote an implementation-defined component (see 13.5.1) or a class-wide type implicitly declared immediately within the same declarative region as the item. A local_name that is a *library_unit_*name (only permitted in a representation pragma) shall resolve to denote the library_item that immediately precedes (except for other pragmas) the representation pragma.

5.a/1 **Reason:** {*8652/0009*} {*AI95-00137-01*} This is a Name Resolution Rule, because we don't want an operational or representation item for X to be ambiguous just because there's another X declared in an outer declarative region. It doesn't make much difference, since most operational or representation items are for types or subtypes, and type and subtype names can't be overloaded.

5.b/1 **Ramification:** {*8652/0009*} {*AI95-00137-01*} The visibility rules imply that the declaration has to occur before the operational or representation item.

5.c/1 {*8652/0009*} {*AI95-00137-01*} For objects, this implies that operational or representation items can be applied only to stand-alone objects.

Legality Rules

6/1 {*8652/0009*} {*AI95-00137-01*} The local_name of an aspect_clause or representation pragma shall statically denote an entity (or, in the case of a pragma, one or more entities) declared immediately preceding it in a compilation, or within the same declarative_part, package_specification, task_definition, protected_definition, or record_definition as the representation or operational item. If a local_name denotes a [local] callable entity, it may do so through a [local] subprogram_renaming_declaration [(as a way to resolve ambiguity in the presence of overloading)]; otherwise, the local_name shall not denote a renaming_declaration.

6.a **Ramification:** The “statically denote” part implies that it is impossible to specify the representation of an object that is not a stand-alone object, except in the case of a representation item like pragma Atomic that is allowed inside a component_list (in which case the representation item specifies the representation of components of all objects of the type). It also prevents the problem of renamings of things like “P.**all**” (where P is an access-to-subprogram value) or “E(I)” (where E is an entry family).

6.b The part about where the denoted entity has to have been declared appears twice — once as a Name Resolution Rule, and once as a Legality Rule. Suppose P renames Q, and we have a representation item in a declarative_part whose local_name is P. The fact that the representation item has to appear in the same declarative_part as P is a Name Resolution Rule, whereas the fact that the representation item has to appear in the same declarative_part as Q is a Legality Rule. This is subtle, but it seems like the least confusing set of rules.

6.c **Discussion:** A separate Legality Rule applies for component_clauses. See 13.5.1, “Record Representation Clauses”.

7/2 {*AI95-00291-02*} {*representation of an object*} {*size (of an object)*} The *representation* of an object consists of a certain number of bits (the *size* of the object). For an object of an elementary type, these are the bits that are normally read or updated by the machine code when loading, storing, or operating-on the value of the object. For an object of a composite type, these are the bits reserved for this object, and include bits occupied by subcomponents of the object. If the size of an object is greater than that of its subtype, the additional bits are padding bits. {*padding bits*} For an elementary object, these padding bits are normally read and updated along with the others. For a composite object, padding bits might not be read or updated in any given composite operation, depending on the implementation.

7.a/2 **To be honest:** {*AI95-00291-02*} {*contiguous representation* [partial]} {*discontiguous representation* [partial]} Discontiguous representations are allowed, but the ones we're interested in here are generally contiguous sequences of bits. For a discontiguous representation, the size doesn't necessarily describe the “footprint” of the object in memory (that is, the amount of space taken in the address space for the object).

7.a.1/2 **Discussion:** {*AI95-00291-02*} In the case of composite objects, we want the implementation to have the flexibility to either do operations component-by-component, or with a block operation covering all of the bits. We carefully avoid giving a preference in the wording. There is no requirement for the choice to be documented, either, as the implementation can make that choice based on many factors, and could make a different choice for different operations on the same object.

{*AI95-00291-02*} In the case of a properly aligned, contiguous object whose size is a multiple of the storage unit size, no other bits should be read or updated as part of operating on the object. We don't say this normatively because it would be difficult to normatively define "properly aligned" or "contiguous". 7.a.2/2

Ramification: Two objects with the same value do not necessarily have the same representation. For example, an implementation might represent False as zero and True as any odd value. Similarly, two objects (of the same type) with the same sequence of bits do not necessarily have the same value. For example, an implementation might use a biased representation in some cases but not others: 7.b

7.c
```
subtype S is Integer range 1..256;
type A is array(Natural range 1..4) of S;
pragma Pack(A);
X : S := 3;
Y : A := (1, 2, 3, 4);
```

The implementation might use a biased-by-1 representation for the array elements, but not for X. X and Y(3) have the same value, but different representation: the representation of X is a sequence of (say) 32 bits: 0...011, whereas the representation of Y(3) is a sequence of 8 bits: 00000010 (assuming a two's complement representation). 7.d

Such tricks are not required, but are allowed. 7.e

Discussion: The value of any padding bits is not specified by the language, though for a numeric type, it will be much harder to properly implement the predefined operations if the padding bits are not either all zero, or a sign extension. 7.f

Ramification: For example, suppose S'Size = 2, and an object X is of subtype S. If the machine code typically uses a 32-bit load instruction to load the value of X, then X'Size should be 32, even though 30 bits of the value are just zeros or sign-extension bits. On the other hand, if the machine code typically masks out those 30 bits, then X'Size should be 2. Usually, such masking only happens for components of a composite type for which packing, Component_Size, or record layout is specified. 7.g

Note, however, that the formal parameter of an instance of Unchecked_Conversion is a special case. Its Size is required to be the same as that of its subtype. 7.h

Note that we don't generally talk about the representation of a value. A value is considered to be an amorphous blob without any particular representation. An object is considered to be more concrete. 7.i

{*aspect of representation* [distributed]} {*representation aspect*} {*directly specified (of an aspect of representation of an entity)*} A representation item *directly specifies* an *aspect of representation* of the entity denoted by the local_name, except in the case of a type-related representation item, whose local_name shall denote a first subtype, and which directly specifies an aspect of the subtype's type. {*type-related (representation item)* [distributed]} {*subtype-specific (of a representation item)* [distributed]} {*type-related (aspect)* [distributed]} {*subtype-specific (of an aspect)* [distributed]} A representation item that names a subtype is either *subtype-specific* (Size and Alignment clauses) or *type-related* (all others). [Subtype-specific aspects may differ for different subtypes of the same type.] 8

To be honest: *Type-related* and *subtype-specific* are defined likewise for the corresponding aspects of representation. 8.a

To be honest: Some representation items directly specify more than one aspect. 8.b

Discussion: For example, a pragma Export specifies the convention of an entity, and also specifies that it is exported. 8.c

Ramification: Each specifiable attribute constitutes a separate aspect. An enumeration_representation_clause specifies the coding aspect. A record_representation_clause (without the mod_clause) specifies the record layout aspect. Each representation pragma specifies a separate aspect. 8.d

Reason: We don't need to say that an at_clause or a mod_clause specify separate aspects, because these are equivalent to attribute_definition_clauses. See J.7, "At Clauses", and J.8, "Mod Clauses". 8.e

Ramification: The following representation items are type-related: 8.f

- enumeration_representation_clause 8.g
- record_representation_clause 8.h
- Component_Size clause 8.i
- *This paragraph was deleted.*{*8652/0009*} {*AI95-00137-01*} 8.j/1
- Small clause 8.k
- Bit_Order clause 8.l
- Storage_Pool clause 8.m

8.n
- Storage_Size clause

8.n.1/2
- {*AI95-00270-01*} Stream_Size clause

8.o/1
- *This paragraph was deleted.*{*8652/0009*} {*AI95-00137-01*}

8.p/1
- *This paragraph was deleted.*{*8652/0009*} {*AI95-00137-01*}

8.q/1
- *This paragraph was deleted.*{*8652/0009*} {*AI95-00137-01*}

8.r/1
- *This paragraph was deleted.*{*8652/0009*} {*AI95-00137-01*}

8.s
- Machine_Radix clause

8.t
- pragma Pack

8.u
- pragmas Import, Export, and Convention (when applied to a type)

8.v
- pragmas Atomic and Volatile (when applied to a type)

8.w
- pragmas Atomic_Components and Volatile_Components (when applied to an array type)

8.x
- pragma Discard_Names (when applied to an enumeration or tagged type)

8.y The following representation items are subtype-specific:

8.z
- Alignment clause (when applied to a first subtype)

8.aa
- Size clause (when applied to a first subtype)

8.bb The following representation items do not apply to subtypes, so they are neither type-related nor subtype-specific:

8.cc
- Address clause (applies to objects and program units)

8.dd
- Alignment clause (when applied to an object)

8.ee
- Size clause (when applied to an object)

8.ff
- pragmas Import, Export, and Convention (when applied to anything other than a type)

8.gg
- pragmas Atomic and Volatile (when applied to an object or a component)

8.hh
- pragmas Atomic_Components and Volatile_Components (when applied to an array object)

8.ii
- pragma Discard_Names (when applied to an exception)

8.jj
- pragma Asynchronous (applies to procedures)

8.kk/2
- {*AI95-00414-01*} pragma No_Return (applies to procedures)

8.1/1 {*8652/0009*} {*AI95-00137-01*} An operational item *directly specifies* an *operational aspect* of the type of the subtype denoted by the local_name. The local_name of an operational item shall denote a first subtype. An operational item that names a subtype is type-related. {*operational aspect* [distributed]} {*directly specified (of an operational aspect of an entity)*} {*type-related (operational item)* [distributed]} {*type-related (aspect)* [partial]}

8.ll/1 **Ramification:** {*8652/0009*} {*AI95-00137-01*} The following operational items are type-related:

8.mm/1
- External_Tag clause

8.nn/1
- Read clause

8.oo/1
- Write clause

8.pp/1
- Input clause

8.qq/1
- Output clause

9 A representation item that directly specifies an aspect of a subtype or type shall appear after the type is completely defined (see 3.11.1), and before the subtype or type is frozen (see 13.14). If a representation item is given that directly specifies an aspect of an entity, then it is illegal to give another representation item that directly specifies the same aspect of the entity.

9.a/1 **Ramification:** {*8652/0009*} {*AI95-00137-01*} The fact that a representation item (or operational item, see next paragraph) that directly specifies an aspect of an entity is required to appear before the entity is frozen prevents changing the representation of an entity after using the entity in ways that require the representation to be known.

9.1/1 {*8652/0009*} {*AI95-00137-01*} An operational item that directly specifies an aspect of a type shall appear before the type is frozen (see 13.14). If an operational item is given that directly specifies an aspect of a type, then it is illegal to give another operational item that directly specifies the same aspect of the type.

Ramification: Unlike representation items, operational items can be specified on partial views. Since they don't affect the representation, the full declaration need not be known to determine their legality. 9.a.1/1

For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms. 10

Ramification: {*8652/0009*} {*AI95-00137-01*} On the other hand, subtype-specific representation items may be given for the first subtype of such a type, as can operational items. 10.a/1

Reason: The reason for forbidding type-related representation items on untagged by-reference types is because a change of representation is impossible when passing by reference (to an inherited subprogram). The reason for forbidding type-related representation items on untagged types with user-defined primitive subprograms was to prevent implicit change of representation for type-related aspects of representation upon calling inherited subprograms, because such changes of representation are likely to be expensive at run time. Changes of subtype-specific representation attributes, however, are likely to be cheap. This rule is not needed for tagged types, because other rules prevent a type-related representation item from changing the representation of the parent part; we want to allow a type-related representation item on a type extension to specify aspects of the extension part. For example, a pragma Pack will cause packing of the extension part, but not of the parent part. 10.b

{*8652/0009*} {*AI95-00137-01*} {*8652/0011*} {*AI95-00117-01*} {*AI95-00326-01*} Operational and representation aspects of a generic formal parameter are the same as those of the actual. Operational and representation aspects are the same for all views of a type. A type-related representation item is not allowed for a descendant of a generic formal untagged type. 11/2

Ramification: {*8652/0009*} {*AI95-00137-01*} Representation items are allowed for types whose subcomponent types or index subtypes are generic formal types. Operational items and subtype-related representation items are allowed on descendants of generic formal types. 11.a/1

Reason: Since it is not known whether a formal type has user-defined primitive subprograms, specifying type-related representation items for them is not allowed, unless they are tagged (in which case only the extension part is affected in any case). 11.b

Ramification: {*AI95-00326-01*} All views of a type, including the incomplete and partial views, have the same operational and representation aspects. That's important so that the properties don't change when changing views. While most aspects are not available for an incomplete view, we don't want to leave any holes by not saying that they are the same. 11.c/2

A representation item that specifies the Size for a given subtype, or the size or storage place for an object (including a component) of a given subtype, shall allow for enough storage space to accommodate any value of the subtype. 12

{*8652/0009*} {*AI95-00137-01*} A representation or operational item that is not supported by the implementation is illegal, or raises an exception at run time. 13/1

{*AI95-00251-01*} A type_declaration is illegal if it has one or more progenitors, and a representation item applies to an ancestor, and this representation item conflicts with the representation of some other ancestor. The cases that cause conflicts are implementation defined. 13.1/2

Implementation defined: The cases that cause conflicts between the representation of the ancestors of a type_declaration. 13.a/2

Reason: This rule is needed because it may be the case that only the combination of types in a type declaration causes a conflict. Thus it is not possible, in general, to reject the original representation item. For instance: 13.b/2

13.c/2
```
package Pkg1 is
   type Ifc is interface;
   type T is tagged record
      Fld : Integer;
   end record;
   for T use record
      Fld at 0 range 0 .. Integer'Size - 1;
   end record;
end Pkg1;
```

Assume the implementation uses a single tag with a default offset of zero, and that it allows the use of non-default locations for the tag (and thus accepts representation items like the one above). The representation item will force a non-default location for the tag (by putting a component other than the tag into the default location). Clearly, this 13.d/2 package will be accepted by the implementation. However, other declarations could cause trouble. For instance, the implementation could reject:

13.e/2
```
with Pkg1;
package Pkg2 is
   type NewT is new Pkg1.T and Pkg1.Ifc with null record;
end Pkg2;
```

13.f/2 because the declarations of T and Ifc have a conflict in their representation items. This is clearly necessary (it's hard to imagine how Ifc'Class could work with the tag at a location other than the one it is expecting).

13.g/2 Conflicts will usually involve implementation-defined attributes (for specifying the location of the tag, for instance), although the example above shows that doesn't have to be the case. For this reason, we didn't try to specify exactly what causes a conflict; it will depend on the implementation's implementation model and what representation items it allows.

13.h/2 **Implementation Note:** An implementation can only use this rule to reject type_declarations where one its ancestors has a representation item. An implementation must ensure that the default representations of ancestors cannot conflict.

Static Semantics

14 If two subtypes statically match, then their subtype-specific aspects (Size and Alignment) are the same. {*statically matching (effect on subtype-specific aspects)* [partial]}

14.a **Reason:** This is necessary because we allow (for example) conversion between access types whose designated subtypes statically match. Note that it is illegal to specify an aspect (including a subtype-specific one) for a nonfirst subtype.

14.b Consider, for example:

14.c/1
```
package P1 is
    subtype S1 is Integer range 0..2**16-1;
    for S1'Size use 16; -- Illegal!
            -- S1'Size would be 16 by default.
    type A1 is access all S1;
    X1: A1;
end P1;
```

14.d/1
```
package P2 is
    subtype S2 is Integer range 0..2**16-1;
    for S2'Size use 32; -- Illegal!
    type A2 is access all S2;
    X2: A2;
end P2;
```

14.e/1
```
procedure Q is
    use P1, P2;
    type Array1 is array(Integer range <>) of aliased S1;
    pragma Pack(Array1);
    Obj1: Array1(1..100);
    type Array2 is array(Integer range <>) of aliased S2;
    pragma Pack(Array2);
    Obj2: Array2(1..100);
begin
    X1 := Obj2(17)'Unchecked_Access;
    X2 := Obj1(17)'Unchecked_Access;
end Q;
```

14.f Loads and stores through X1 would read and write 16 bits, but X1 points to a 32-bit location. Depending on the endianness of the machine, loads might load the wrong 16 bits. Stores would fail to zero the other half in any case.

14.g Loads and stores through X2 would read and write 32 bits, but X2 points to a 16-bit location. Thus, adjacent memory locations would be trashed.

14.h Hence, the above is illegal. Furthermore, the compiler is forbidden from choosing different Sizes by default, for the same reason.

14.i The same issues apply to Alignment.

15/1 {*8652/0040*} {*AI95-00108-01*} A derived type inherits each type-related aspect of representation of its parent type that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent type from the grandparent type. A derived subtype inherits each subtype-specific aspect of representation of its parent subtype that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent subtype from the grandparent subtype, but only if the parent subtype statically matches the first subtype of the parent type. An inherited aspect of representation is overridden by a subsequent representation item that specifies the same aspect of the type or subtype.

15.a **To be honest:** A record_representation_clause for a record extension does not override the layout of the parent part; if the layout was specified for the parent type, it is inherited by the record extension.

15.b **Ramification:** If a representation item for the parent appears after the derived_type_definition, then inheritance does not happen for that representation item.

15.1/2 {*8652/0040*} {*AI95-00108-01*} {*AI95-00444-01*} In contrast, whether operational aspects are inherited by an untagged derived type depends on each specific aspect. [Operational aspects are never inherited for a tagged type.] When operational aspects are inherited by an untagged derived type, aspects that were directly specified by operational items that are visible at the point of the derived type declaration, or (in the case where the parent is derived) that were inherited by the parent type from the grandparent type are inherited. An inherited operational aspect is overridden by a subsequent operational item that specifies the same aspect of the type.

15.b.1/1 **Ramification:** As with representation items, if an operational item for the parent appears after the derived_type_definition, then inheritance does not happen for that operational item.

15.b.2/2 **Discussion:** {*AI95-00444-01*} Only untagged types inherit operational aspects. Inheritance from tagged types causes problems, as the different views can have different visibility on operational items — potentially leading to operational items that depend on the view. We want aspects to be the same for all views. Untagged types don't have this problem as plain private types don't have ancestors, and thus can't inherit anything. In addition, it seems unlikely that we'll need inheritance for tagged types, as usually we'll want to incorporate the parent's operation into a new one that also handles any extension components.

15.2/2 {*AI95-00444-01*} When an aspect that is a subprogram is inherited, the derived type inherits the aspect in the same way that a derived type inherits a user-defined primitive subprogram from its parent (see 3.4).

15.c/2 **Reason:** This defines the parameter names and types, and the needed implicit conversions.

16 Each aspect of representation of an entity is as follows:

17
- {*specified (of an aspect of representation of an entity)*} If the aspect is *specified* for the entity, meaning that it is either directly specified or inherited, then that aspect of the entity is as specified, except in the case of Storage_Size, which specifies a minimum.

17.a **Ramification:** This rule implies that queries of the aspect return the specified value. For example, if the user writes "**for** X'Size **use** 32;", then a query of X'Size will return 32.

18
- {*unspecified* [partial]} If an aspect of representation of an entity is not specified, it is chosen by default in an unspecified manner.

18.a/1 **Ramification:** {*8652/0009*} {*AI95-00137-01*} Note that representation items can affect the semantics of the entity.

18.b The rules forbid things like "**for** S'Base'Alignment **use** ..." and "**for** S'Base **use** record ...".

18.c **Discussion:** The intent is that implementations will represent the components of a composite value in the same way for all subtypes of a given composite type. Hence, Component_Size and record layout are type-related aspects.

18.1/1 {*8652/0040*} {*AI95-00108-01*} {*specified (of an operational aspect of an entity)*} If an operational aspect is *specified* for an entity (meaning that it is either directly specified or inherited), then that aspect of the entity is as specified. Otherwise, the aspect of the entity has the default value for that aspect.

18.2/2 {*AI95-00291-02*} A representation item that specifies an aspect of representation that would have been chosen in the absence of the representation item is said to be *confirming*.{*confirming (representation item)*}

Dynamic Semantics

19/1 {*8652/0009*} {*AI95-00137-01*} {*elaboration (aspect_clause)* [partial]} For the elaboration of an aspect_clause, any evaluable constructs within it are evaluated.

19.a **Ramification:** Elaboration of representation pragmas is covered by the general rules for pragmas in Section 2.

Implementation Permissions

20 An implementation may interpret aspects of representation in an implementation-defined manner. An implementation may place implementation-defined restrictions on representation items. {*recommended level of support* [distributed]} A *recommended level of support* is specified for representation items and related features in each subclause. These recommendations are changed to requirements for implementations that support the Systems Programming Annex (see C.2, “Required Representation Support”).

20.a **Implementation defined:** The interpretation of each aspect of representation.

20.b **Implementation defined:** Any restrictions placed upon representation items.

20.c **Ramification:** Implementation-defined restrictions may be enforced either at compile time or at run time. There is no requirement that an implementation justify any such restrictions. They can be based on avoiding implementation complexity, or on avoiding excessive inefficiency, for example.

20.c.1/1 {*8652/0009*} {*AI95-00137-01*} There is no such permission for operational aspects.

Implementation Advice

21 {*recommended level of support (with respect to nonstatic expressions)* [partial]} The recommended level of support for all representation items is qualified as follows:

21.1/2 • {*AI95-00291-02*} A confirming representation item should be supported.

21.a.1/2 **To be honest:** A confirming representation item might not be possible for some entities. For instance, consider an unconstrained array. The size of such a type is implementation-defined, and might not actually be a representable value, or might not be static.

22 • An implementation need not support representation items containing nonstatic expressions, except that an implementation should support a representation item for a given entity if each nonstatic expression in the representation item is a name that statically denotes a constant declared before the entity.

22.a **Reason:** This is to avoid the following sort of thing:

22.b
```
X : Integer := F(...);
Y : Address := G(...);
for X'Address use Y;
```

22.c In the above, we have to evaluate the initialization expression for X before we know where to put the result. This seems like an unreasonable implementation burden.

22.d The above code should instead be written like this:

22.e
```
Y : constant Address := G(...);
X : Integer := F(...);
for X'Address use Y;
```

22.f This allows the expression “Y” to be safely evaluated before X is created.

22.g The constant could be a formal parameter of mode **in**.

22.h An implementation can support other nonstatic expressions if it wants to. Expressions of type Address are hardly ever static, but their value might be known at compile time anyway in many cases.

23 • An implementation need not support a specification for the Size for a given composite subtype, nor the size or storage place for an object (including a component) of a given composite subtype, unless the constraints on the subtype and its composite subcomponents (if any) are all static constraints.

24/2 • {*AI95-00291-02*} An implementation need not support a nonconfirming representation item if it could cause an aliased object or an object of a by-reference type to be allocated at a nonaddressable location or, when the alignment attribute of the subtype of such an object is nonzero, at an address that is not an integral multiple of that alignment.

24.a/1 **Reason:** The intent is that access types, type System.Address, and the pointer used for a by-reference parameter should be implementable as a single machine address — bit-field pointers should not be required. (There is no requirement that this implementation be used — we just want to make sure it's feasible.)

24.b/2 **Implementation Note:** {*AI95-00291-02*} We want subprograms to be able to assume the properties of the types of their parameters inside of subprograms. While many objects can be copied to allow this (and thus do not need limitations), aliased or by-reference objects cannot be copied (their memory location is part of their identity). Thus, the above rule does not apply to types that merely allow by-reference parameter passing; for such types, a copy typically needs to be made at the call site when a bit-aligned component is passed as a parameter.

- 25/2 {*AI95-00291-02*} An implementation need not support a nonconfirming representation item if it could cause an aliased object of an elementary type to have a size other than that which would have been chosen by default.

25.a/2 **Reason:** Since all bits of elementary objects participate in operations, aliased objects must not have a different size than that assumed by users of the access type.

- 26/2 {*AI95-00291-02*} An implementation need not support a nonconfirming representation item if it could cause an aliased object of a composite type, or an object whose type is by-reference, to have a size smaller than that which would have been chosen by default.

26.a/2 **Reason:** Unlike elementary objects, there is no requirement that all bits of a composite object participate in operations. Thus, as long as the object is the same or larger in size than that expected by the access type, all is well.

26.b/2 **Ramification:** This rule presumes that the implementation allocates an object of a size specified to be larger than the default size in such a way that access of the default size suffices to correctly read and write the value of the object.

- 27/2 {*AI95-00291-02*} An implementation need not support a nonconfirming subtype-specific representation item specifying an aspect of representation of an indefinite or abstract subtype.

27.a/2 **Reason:** Aspects of representations are often not well-defined for such types.

27.b/2 **Ramification:** {*AI95-00291-02*} A pragma Pack will typically not pack so tightly as to disobey the above rules. A Component_Size clause or record_representation_clause will typically be illegal if it disobeys the above rules. Atomic components have similar restrictions (see C.6, “Shared Variable Control”).

28/2 {*AI95-00291-02*} For purposes of these rules, the determination of whether a representation item applied to a type *could cause* an object to have some property is based solely on the properties of the type itself, not on any available information about how the type is used. In particular, it presumes that minimally aligned objects of this type might be declared at some point.

28.a/2 **Implementation Advice:** The recommended level of support for all representation items should be followed.

Incompatibilities With Ada 83

28.b {*incompatibilities with Ada 83*} It is now illegal for a representation item to cause a derived by-reference type to have a different record layout from its parent. This is necessary for by-reference parameter passing to be feasible. This only affects programs that specify the representation of types derived from types containing tasks; most by-reference types are new to Ada 95. For example, if A1 is an array of tasks, and A2 is derived from A1, it is illegal to apply a pragma Pack to A2.

Extensions to Ada 83

28.c/1 {*8652/0009*} {*AI95-00137-01*} {*extensions to Ada 83*} Ada 95 allows additional aspect_clauses for objects.

Wording Changes from Ada 83

28.d/1 {*8652/0009*} {*AI95-00137-01*} The syntax rule for type_representation_clause is removed; the right-hand side of that rule is moved up to where it was used, in aspect_clause. There are two references to “type representation clause” in RM83, both in Section 13; these have been reworded. Also, the representation_clause has been renamed the aspect_clause to reflect that it can be used to control more than just representation aspects.

28.e/2 {*8652/0009*} {*AI95-00137-01*} {*AI95-00114-01*} We have defined a new term “representation item,” which includes all representation clauses and representation pragmas, as well as component_clauses. This is convenient because the rules are almost identical for all of them. We have also defined the new terms “operational item” and “operational aspects” in order to conveniently handle new types of specifiable entities.

28.f All of the forcing occurrence stuff has been moved into its own subclause (see 13.14), and rewritten to use the term “freezing”.

28.g RM83-13.1(10) requires implementation-defined restrictions on representation items to be enforced at compile time. However, that is impossible in some cases. If the user specifies a junk (nonstatic) address in an address clause, and the implementation chooses to detect the error (for example, using hardware memory management with protected pages), then it's clearly going to be a run-time error. It seems silly to call that "semantics" rather than "a restriction."

28.h RM83-13.1(10) tries to pretend that representation_clauses don't affect the semantics of the program. One counter-example is the Small clause. Ada 95 has more counter-examples. We have noted the opposite above.

28.i Some of the more stringent requirements are moved to C.2, "Required Representation Support".

Extensions to Ada 95

28.j/2 {*AI95-00291-02*} {*extensions to Ada 95*} **Amendment Correction:** Confirming representation items are defined, and the recommended level of support is now that they always be supported.

Wording Changes from Ada 95

28.k/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added operational items in order to eliminate unnecessary restrictions and permissions on stream attributes. As part of this, representation_clause was renamed to aspect_clause.

28.l/2 {*8652/0009*} {*AI95-00137-01*} {*AI95-00326-01*} **Corrigendum:** Added wording to say that the partial and full views have the same operational and representation aspects. Ada 2005 extends this to cover all views, including the incomplete view.

28.m/2 {*8652/0040*} {*AI95-00108-01*} **Corrigendum:** Changed operational items to have inheritance specified for each such aspect.

28.n/2 {*AI95-00251-01*} Added wording to allow the rejection of types with progenitors that have conflicting representation items.

28.o/2 {*AI95-00291-02*} The description of the representation of an object was clarified (with great difficulty reaching agreement). Added wording to say that representation items on aliased and by-reference objects never need be supported if they would not be implementable without distributed overhead even if other recommended level of support says otherwise. This wording matches the rules with reality.

28.p/2 {*AI95-00444-01*} Added wording so that inheritance depends on whether operational items are visible rather than whether they occur before the declaration (we don't want to look into private parts). Limited operational inheritance to untagged types to avoid anomolies with private extensions (this is not incompatible, no existing operational attribute used this capability). Also added wording to clearly define that subprogram inheritance works like derivation of subprograms.

13.2 Pragma Pack

1 [A pragma Pack specifies that storage minimization should be the main criterion when selecting the representation of a composite type.]

Syntax

2 The form of a pragma Pack is as follows:

3 **pragma** Pack(*first_subtype*_local_name);

Legality Rules

4 The *first_subtype*_local_name of a pragma Pack shall denote a composite subtype.

Static Semantics

5 {*representation pragma (Pack)* [partial]} {*pragma, representation (Pack)* [partial]} {*aspect of representation (packing)* [partial]} {*packing (aspect of representation)*} {*packed*} A pragma Pack specifies the *packing* aspect of representation; the type (or the extension part) is said to be *packed*. For a type extension, the parent part is packed as for the parent type, and a pragma Pack causes packing only of the extension part.

5.a **Ramification:** The only high level semantic effect of a pragma Pack is independent addressability (see 9.10, "Shared Variables").

Implementation Advice

6 If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations.

6.a.1/2 **Implementation Advice:** Storage allocated to objects of a packed type should be minimized.

6.a **Ramification:** A pragma Pack is for gaining space efficiency, possibly at the expense of time. If more explicit control over representation is desired, then a record_representation_clause, a Component_Size clause, or a Size clause should be used instead of, or in addition to, a pragma Pack.

6.1/2 {*AI95-00291-02*} If a packed type has a component that is not of a by-reference type and has no aliased part, then such a component need not be aligned according to the Alignment of its subtype; in particular it need not be allocated on a storage element boundary.

7 {*recommended level of support (pragma Pack)* [partial]} The recommended level of support for pragma Pack is:

- 8 For a packed record type, the components should be packed as tightly as possible subject to the Sizes of the component subtypes, and subject to any record_representation_clause that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words.

8.a **Ramification:** The implementation can always allocate an integral number of words for a component that will not fit in a word. The rule also allows small component sizes to be rounded up if such rounding does not waste space. For example, if Storage_Unit = 8, then a component of size 8 is probably more efficient than a component of size 7 plus a 1-bit gap (assuming the gap is needed anyway).

- 9 For a packed array type, if the component subtype's Size is less than or equal to the word size, and Component_Size is not specified for the type, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size.

9.a **Ramification:** If a component subtype is aliased, its Size will generally be a multiple of Storage_Unit, so it probably won't get packed very tightly.

9.b/2 **Implementation Advice:** The recommended level of support for pragma Pack should be followed.

Wording Changes from Ada 95

9.c/2 {*AI95-00291-02*} Added clarification that pragma Pack can ignore alignment requirements on types that don't have by-reference or aliased parts. This was always intended, but there was no wording to that effect.

13.3 Operational and Representation Attributes

1/1 {*8652/0009*} {*AI95-00137-01*} [{*representation attribute*} {*attribute (representation)*} The values of certain implementation-dependent characteristics can be obtained by interrogating appropriate operational or representation attributes. {*attribute (specifying)* [distributed]} Some of these attributes are specifiable via an attribute_definition_clause.]

Language Design Principles

1.a In general, the meaning of a given attribute should not depend on whether the attribute was specified via an attribute_definition_clause, or chosen by default by the implementation.

Syntax

2
```
attribute_definition_clause ::=
    for local_name'attribute_designator use expression;
  | for local_name'attribute_designator use name;
```

Name Resolution Rules

3 For an attribute_definition_clause that specifies an attribute that denotes a value, the form with an expression shall be used. Otherwise, the form with a name shall be used.

4 {*expected type (attribute_definition_clause expression or name)* [partial]} For an attribute_definition_clause that specifies an attribute that denotes a value or an object, the expected type for the expression or name is that of the attribute. {*expected profile (attribute_definition_clause name)* [partial]} For an attribute_definition_clause that specifies an attribute that denotes a subprogram, the expected profile for the name is the profile required for the attribute. For an attribute_definition_clause that specifies an attribute that denotes some other kind of entity, the name shall resolve to denote an entity of the appropriate kind.

4.a **Ramification:** For example, the Size attribute is of type *universal_integer*. Therefore, the expected type for Y in "**for** X'Size **use** Y;" is *universal_integer*, which means that Y can be of any integer type.

4.b **Discussion:** For attributes that denote subprograms, the required profile is indicated separately for the individual attributes.

4.c **Ramification:** For an attribute_definition_clause with a name, the name need not statically denote the entity it denotes. For example, the following kinds of things are allowed:

4.d
```
    for Some_Access_Type'Storage_Pool use Storage_Pool_Array(I);
    for Some_Type'Read use Subprogram_Pointer.all;
```

Legality Rules

5/1 {*8652/0009*} {*AI95-00137-01*} {*specifiable (of an attribute and for an entity)* [distributed]} {*attribute (specifiable)* [distributed]} An attribute_designator is allowed in an attribute_definition_clause only if this International Standard explicitly allows it, or for an implementation-defined attribute if the implementation allows it. {*aspect of representation (specifiable attributes)* [partial]} Each specifiable attribute constitutes an {*operational aspect (specifiable attributes)* [partial]} operational aspect or aspect of representation.

5.a **Discussion:** For each specifiable attribute, we generally say something like, "The ... attribute may be specified for ... via an attribute_definition_clause."

5.b The above wording allows for T'Class'Alignment, T'Class'Size, T'Class'Input, and T'Class'Output to be specifiable.

5.c A specifiable attribute is not necessarily specifiable for all entities for which it is defined. For example, one is allowed to ask T'Component_Size for an array subtype T, but "**for** T'Component_Size **use** ..." is only allowed if T is a first subtype, because Component_Size is a type-related aspect.

6 For an attribute_definition_clause that specifies an attribute that denotes a subprogram, the profile shall be mode conformant with the one required for the attribute, and the convention shall be Ada. Additional requirements are defined for particular attributes. {*mode conformance (required)*}

6.a **Ramification:** This implies, for example, that if one writes:

6.b
```
    for T'Read use R;
```

6.c R has to be a procedure with two parameters with the appropriate subtypes and modes as shown in 13.13.2.

Static Semantics

7/2 {*AI95-00270-01*} {*Address clause*} {*Alignment clause*} {*Size clause*} {*Component_Size clause*} {*External_Tag clause*} {*Small clause*} {*Bit_Order clause*} {*Storage_Pool clause*} {*Storage_Size clause*} {*Stream_Size clause*} {*Read clause*} {*Write clause*} {*Input clause*} {*Output clause*} {*Machine_Radix clause*} A *Size clause* is an attribute_definition_clause whose attribute_designator is Size. Similar definitions apply to the other specifiable attributes.

7.a **To be honest:** {*type-related (attribute_definition_clause)* [partial]} {*subtype-specific (attribute_definition_clause)* [partial]} An attribute_definition_clause is type-related or subtype-specific if the attribute_designator denotes a type-related or subtype-specific attribute, respectively.

{*storage element*} {*byte: See storage element*} A *storage element* is an addressable element of storage in the machine. {*word*} A *word* is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of an integral number of storage elements. 8

Discussion: A storage element is not intended to be a single bit, unless the machine can efficiently address individual bits. 8.a

Ramification: For example, on a machine with 8-bit storage elements, if there exist 32-bit integer registers, with a full set of arithmetic and logical instructions to manipulate those registers, a word ought to be 4 storage elements — that is, 32 bits. 8.b

Discussion: The "given the implementation's run-time model" part is intended to imply that, for example, on an 80386 running MS-DOS, the word might be 16 bits, even though the hardware can support 32 bits. 8.c

A word is what ACID refers to as a "natural hardware boundary". 8.d

Storage elements may, but need not be, independently addressable (see 9.10, "Shared Variables"). Words are expected to be independently addressable. 8.e

{*AI95-00133-01*} {*machine scalar*} A *machine scalar* is an amount of storage that can be conveniently and efficiently loaded, stored, or operated upon by the hardware. Machine scalars consist of an integral number of storage elements. The set of machine scalars is implementation defined, but must include at least the storage element and the word. Machine scalars are used to interpret component_clauses when the nondefault bit ordering applies. 8.1/2

Implementation defined: The set of machine scalars. 8.e.1/2

{*8652/0009*} {*AI95-00137-01*} The following representation attributes are defined: Address, Alignment, Size, Storage_Size, and Component_Size. 9/1

For a prefix X that denotes an object, program unit, or label: 10/1

X'Address Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address. 11

Ramification: Here, the "first of the storage elements" is intended to mean the one with the lowest address; the endianness of the machine doesn't matter. 11.a

{*specifiable (of Address for stand-alone objects and for program units)* [partial]} {*Address clause*} Address may be specified for stand-alone objects and for program units via an attribute_definition_clause. 12

Ramification: Address is not allowed for enumeration literals, predefined operators, derived task types, or derived protected types, since they are not program units. 12.a

The validity of a given address depends on the run-time model; thus, in order to use Address clauses correctly, one needs intimate knowledge of the run-time model. 12.b

If the Address of an object is specified, any explicit or implicit initialization takes place as usual, unless a pragma Import is also specified for the object (in which case any necessary initialization is presumably done in the foreign language). 12.c

Any compilation unit containing an attribute_reference of a given type depends semantically on the declaration of the package in which the type is declared, even if not mentioned in an applicable with_clause — see 10.1.1. In this case, it means that if a compilation unit contains X'Address, then it depends on the declaration of System. Otherwise, the fact that the value of Address is of a type in System wouldn't make sense; it would violate the "legality determinable via semantic dependences" Language Design Principle. 12.d

AI83-00305 — If X is a task type, then within the body of X, X denotes the current task object; thus, X'Address denotes the object's address. 12.e

Interrupt entries and their addresses are described in J.7.1, "Interrupt Entries". 12.f

If X is not allocated on a storage element boundary, X'Address points at the first of the storage elements that contains any part of X. This is important for the definition of the Position attribute to be sensible. 12.g

Erroneous Execution

13 {*erroneous execution (cause)* [partial]} If an Address is specified, it is the programmer's responsibility to ensure that the address is valid; otherwise, program execution is erroneous.

Implementation Advice

14 For an array X, X'Address should point at the first component of the array, and not at the array bounds.

14.a.1/2 **Implementation Advice:** For an array X, X'Address should point at the first component of the array rather than the array bounds.

14.a **Ramification:** On the other hand, we have no advice to offer about discriminants and tag fields; whether or not the address points at them is not specified by the language. If discriminants are stored separately, then the Position of a discriminant might be negative, or might raise an exception.

15 {*recommended level of support (Address attribute)* [partial]} The recommended level of support for the Address attribute is:

16 • X'Address should produce a useful result if X is an object that is aliased or of a by-reference type, or is an entity whose Address has been specified.

16.a **Reason:** Aliased objects are the ones for which the Unchecked_Access attribute is allowed; hence, these have to be allocated on an addressable boundary anyway. Similar considerations apply to objects of a by-reference type.

16.b An implementation need not go to any trouble to make Address work in other cases. For example, if an object X is not aliased and not of a by-reference type, and the implementation chooses to store it in a register, X'Address might return System.Null_Address (assuming registers are not addressable). For a subprogram whose calling convention is Intrinsic, or for a package, the implementation need not generate an out-of-line piece of code for it.

17 • An implementation should support Address clauses for imported subprograms.

18/2 • *This paragraph was deleted.*{*AI95-00291-02*}

18.a/2 *This paragraph was deleted.*

19 • If the Address of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases.

19.a/2 **Implementation Advice:** The recommended level of support for the Address attribute should be followed.

NOTES

20 1 The specification of a link name in a pragma Export (see B.1) for a subprogram or object is an alternative to explicit specification of its link-time address, allowing a link-time directive to place the subprogram or object within memory.

21 2 The rules for the Size attribute imply, for an aliased object X, that if X'Size = Storage_Unit, then X'Address points at a storage element containing all of the bits of X, and only the bits of X.

Wording Changes from Ada 83

21.a The intended meaning of the various attributes, and their attribute_definition_clauses, is more explicit.

21.b The address_clause has been renamed to at_clause and moved to Annex J, “Obsolescent Features”. One can use an Address clause (“for T'Address **use** ...;”) instead.

21.c The attributes defined in RM83-13.7.3 are moved to Annex G, A.5.3, and A.5.4.

Language Design Principles

21.d By default, the Alignment of a subtype should reflect the “natural” alignment for objects of the subtype on the machine. The Alignment, whether specified or default, should be known at compile time, even though Addresses are generally not known at compile time. (The generated code should never need to check at run time the number of zero bits at the end of an address to determine an alignment).

21.e There are two symmetric purposes of Alignment clauses, depending on whether or not the implementation has control over object allocation. If the implementation allocates an object, the implementation should ensure that the Address and Alignment are consistent with each other. If something outside the implementation allocates an object, the implementation should be allowed to assume that the Address and Alignment are consistent, but should not assume stricter alignments than that.

Static Semantics

{*AI95-00291-02*} For a prefix X that denotes an object: 22/2

X'Alignment 23/2

{*AI95-00291-02*} The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If X'Alignment is not zero, then X is aligned on a storage unit boundary and X'Address is an integral multiple of X'Alignment (that is, the Address modulo the Alignment is zero).

This paragraph was deleted.{*AI95-00291-02*} 24/2

Ramification: The Alignment is passed by an allocator to the Allocate operation; the implementation has to choose a value such that if the address returned by Allocate is aligned as requested, the generated code can correctly access the object. 24.a

The above mention of “modulo” is referring to the "**mod**" operator declared in System.Storage_Elements; if X **mod** N = 0, then X is by definition aligned on an N-storage-element boundary. 24.b

{*AI95-00291-02*} {*specifiable (of Alignment for objects)* [partial]} {*Alignment clause*} Alignment may be specified for [stand-alone] objects via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. 25/2

This paragraph was deleted.{*AI95-00247-01*} 26/2

{*AI95-00291-02*} For every subtype S: 26.1/2

S'Alignment 26.2/2

{*AI95-00291-02*} The value of this attribute is of type *universal_integer*, and nonnegative.

{*AI95-00051-02*} {*AI95-00291-02*} For an object X of subtype S, if S'Alignment is not zero, then X'Alignment is a nonzero integral multiple of S'Alignment unless specified otherwise by a representation item. 26.3/2

{*AI95-00291-02*} {*specifiable (of Alignment for first subtypes)* [partial]} {*Alignment clause*} Alignment may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. 26.4/2

Erroneous Execution

{*erroneous execution (cause)* [partial]} Program execution is erroneous if an Address clause is given that conflicts with the Alignment. 27

Ramification: The user has to either give an Alignment clause also, or else know what Alignment the implementation will choose by default. 27.a

{*AI95-00051-02*} {*AI95-00291-02*} {*erroneous execution (cause)* [partial]} For an object that is not allocated under control of the implementation, execution is erroneous if the object is not aligned according to its Alignment. 28/2

Implementation Advice

{*recommended level of support (Alignment attribute for subtypes)* [partial]} The recommended level of support for the Alignment attribute for subtypes is: 29

- {*AI95-00051-02*} An implementation should support an Alignment clause for a discrete type, fixed point type, record type, or array type, specifying an Alignment value that is zero or a power of two, subject to the following: 30/2
- {*AI95-00051-02*} An implementation need not support an Alignment clause for a signed integer type specifying an Alignment greater than the largest Alignment value that is ever chosen by default by the implementation for any signed integer type. A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types. 31/2

32/2 • {*AI95-00051-02*} An implementation need not support a nonconfirming Alignment clause which could enable the creation of an object of an elementary type which cannot be easily loaded and stored by available machine instructions.

32.1/2 • {*AI95-00291-02*} An implementation need not support an Alignment specified for a derived tagged type which is not a multiple of the Alignment of the parent type. An implementation need not support a nonconfirming Alignment specified for a derived untagged by-reference type.

32.a/2 **Ramification:** {*AI95-00291-02*} There is no recommendation to support any nonconfirming Alignment clauses for types not mentioned above. Remember that 13.1 requires support for confirming Alignment clauses for all types.

33 {*recommended level of support (Alignment attribute for objects)* [partial]} The recommended level of support for the Alignment attribute for objects is:

34/2 • *This paragraph was deleted.*{*AI95-00291-02*}

35 • For stand-alone library-level objects of statically constrained subtypes, the implementation should support all Alignments supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.

35.1/2 • {*AI95-00291-02*} For other objects, an implementation should at least support the alignments supported for their subtype, subject to the following:

35.2/2 • {*AI95-00291-02*} An implementation need not support Alignments specified for objects of a by-reference type or for objects of types containing aliased subcomponents if the specified Alignment is not a multiple of the Alignment of the subtype of the object.

35.a/2 **Implementation Advice:** The recommended level of support for the Alignment attribute should be followed.

NOTES

36 3 Alignment is a subtype-specific attribute.

37/2 *This paragraph was deleted.*{*AI95-00247-01*}

37.a/2 *This paragraph was deleted.*

38 4 A component_clause, Component_Size clause, or a pragma Pack can override a specified Alignment.

38.a **Discussion:** Most objects are allocated by the implementation; for these, the implementation obeys the Alignment. The implementation is of course allowed to make an object *more* aligned than its Alignment requires — an object whose Alignment is 4 might just happen to land at an address that's a multiple of 4096. For formal parameters, the implementation might want to force an Alignment stricter than the parameter's subtype. For example, on some systems, it is customary to always align parameters to 4 storage elements.

38.b Hence, one might initially assume that the implementation could evilly make all Alignments 1 by default, even though integers, say, are normally aligned on a 4-storage-element boundary. However, the implementation cannot get away with that — if the Alignment is 1, the generated code cannot assume an Alignment of 4, at least not for objects allocated outside the control of the implementation.

38.c Of course implementations can assume anything they can prove, but typically an implementation will be unable to prove much about the alignment of, say, an imported object. Furthermore, the information about where an address “came from” can be lost to the compiler due to separate compilation.

38.d/2 {*AI95-00114-01*} The Alignment of an object that is a component of a packed composite object will usually be 0, to indicate that the component is not necessarily aligned on a storage element boundary. For a subtype, an Alignment of 0 means that objects of the subtype are not normally aligned on a storage element boundary at all. For example, an implementation might choose to make Component_Size be 1 for an array of Booleans, even when pragma Pack has not been specified for the array. In this case, Boolean'Alignment would be 0. (In the presence of tasking, this would in general be feasible only on a machine that had atomic test-bit and set-bit instructions.)

38.e If the machine has no particular natural alignments, then all subtype Alignments will probably be 1 by default.

38.f Specifying an Alignment of 0 in an attribute_definition_clause does not require the implementation to do anything (except return 0 when the Alignment is queried). However, it might be taken as advice on some implementations.

38.g It is an error for an Address clause to disobey the object's Alignment. The error cannot be detected at compile time, in general, because the Address is not necessarily known at compile time (and is almost certainly not static). We do not require a run-time check, since efficiency seems paramount here, and Address clauses are treading on thin ice anyway. Hence, this misuse of Address clauses is just like any other misuse of Address clauses — it's erroneous.

A type extension can have a stricter Alignment than its parent. This can happen, for example, if the Alignment of the parent is 4, but the extension contains a component with Alignment 8. The Alignment of a class-wide type or object will have to be the maximum possible Alignment of any extension. 38.h

The recommended level of support for the Alignment attribute is intended to reflect a minimum useful set of capabilities. An implementation can assume that all Alignments are multiples of each other — 1, 2, 4, and 8 might be the only supported Alignments for subtypes. An Alignment of 3 or 6 is unlikely to be useful. For objects that can be allocated statically, we recommend that the implementation support larger alignments, such as 4096. We do not recommend such large alignments for subtypes, because the maximum subtype alignment will also have to be used as the alignment of stack frames, heap objects, and class-wide objects. Similarly, we do not recommend such large alignments for stack-allocated objects. 38.i

If the maximum default Alignment is 8 (say, Long_Float'Alignment = 8), then the implementation can refuse to accept stricter alignments for subtypes. This simplifies the generated code, since the compiler can align the stack and class-wide types to this maximum without a substantial waste of space (or time). 38.j

Note that the recommended level of support takes into account interactions between Size and Alignment. For example, on a 32-bit machine with 8-bit storage elements, where load and store instructions have to be aligned according to the size of the thing being loaded or stored, the implementation might accept an Alignment of 1 if the Size is 8, but might reject an Alignment of 1 if the Size is 32. On a machine where unaligned loads and stores are merely inefficient (as opposed to causing hardware traps), we would expect an Alignment of 1 to be supported for any Size. 38.k

Wording Changes from Ada 83

The nonnegative part is missing from RM83 (for mod_clauses, nee alignment_clauses, which are an obsolete version of Alignment clauses). 38.l

Static Semantics

For a prefix X that denotes an object: 39/1

X'Size — Denotes the size in bits of the representation of the object. The value of this attribute is of the type *universal_integer*. 40

Ramification: Note that Size is in bits even if Machine_Radix is 10. Each decimal digit (and the sign) is presumably represented as some number of bits. 40.a

{*specifiable (of Size for stand-alone objects)* [partial]} {*Size clause*} Size may be specified for [stand-alone] objects via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative. 41

Implementation Advice

{*AI95-00051-02*} The size of an array object should not include its bounds. 41.1/2

Implementation Advice: The Size of an array object should not include its bounds. 41.a.1/2

{*AI95-00051-02*} {*AI95-00291-02*} {*recommended level of support (Size attribute)* [partial]} The recommended level of support for the Size attribute of objects is the same as for subtypes (see below), except that only a confirming Size clause need be supported for an aliased elementary object. 42/2

- *This paragraph was deleted.*{*AI95-00051-02*} 43/2

Static Semantics

For every subtype S: 44

S'Size — If S is definite, denotes the size [(in bits)] that the implementation would choose for the following objects of subtype S: 45

- A record component of subtype S when the record type is packed. 46
- The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype. 47

If S is indefinite, the meaning is implementation defined. The value of this attribute is of the type *universal_integer*. {*specifiable (of Size for first subtypes)* [partial]} {*Size clause*} The Size of an object is at least as large as that of its subtype, unless the object's Size is determined by a 48

Size clause, a component_clause, or a Component_Size clause. Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative.

48.a **Implementation defined:** The meaning of Size for indefinite subtypes.

48.b **Reason:** The effects of specifying the Size of a subtype are:

- 48.c Unchecked_Conversion works in a predictable manner.
- 48.d A composite type cannot be packed so tightly as to override the specified Size of a component's subtype.
- 48.e Assuming the Implementation Advice is obeyed, if the specified Size allows independent addressability, then the Size of certain objects of the subtype should be equal to the subtype's Size. This applies to stand-alone objects and to components (unless a component_clause or a Component_Size clause applies).

48.f A component_clause or a Component_Size clause can cause an object to be smaller than its subtype's specified size. A pragma Pack cannot; if a component subtype's size is specified, this limits how tightly the composite object can be packed.

48.g The Size of a class-wide (tagged) subtype is unspecified, because it's not clear what it should mean; it should certainly not depend on all of the descendants that happen to exist in a given program. Note that this cannot be detected at compile time, because in a generic unit, it is not necessarily known whether a given subtype is class-wide. It might raise an exception on some implementations.

48.h **Ramification:** A Size clause for a numeric subtype need not affect the underlying numeric type. For example, if I say:

48.i
```
type S is range 1..2;
for S'Size use 64;
```

48.j I am not guaranteed that S'Base'Last >= 2**63–1, nor that intermediate results will be represented in 64 bits.

48.k **Reason:** There is no need to complicate implementations for this sort of thing, because the right way to affect the base range of a type is to use the normal way of declaring the base range:

48.l
```
type Big is range -2**63 .. 2**63 - 1;
subtype Small is Big range 1..1000;
```

48.m **Ramification:** The Size of a large unconstrained subtype (e.g. String'Size) is likely to raise Constraint_Error, since it is a nonstatic expression of type *universal_integer* that might overflow the largest signed integer type. There is no requirement that the largest integer type be able to represent the size in bits of the largest possible object.

Implementation Requirements

49 In an implementation, Boolean'Size shall be 1.

Implementation Advice

50/2 {*AI95-00051-02*} If the Size of a subtype allows for efficient independent addressability (see 9.10) on the target architecture, then the Size of the following objects of the subtype should equal the Size of the subtype:

- 51 Aliased objects (including components).
- 52 Unaliased components, unless the Size of the component is determined by a component_clause or Component_Size clause.

52.a.1/2 **Implementation Advice:** If the Size of a subtype allows for efficient independent addressability, then the Size of most objects of the subtype should equal the Size of the subtype.

52.a **Ramification:** Thus, on a typical 32-bit machine, "**for** S'Size **use** 32;" will guarantee that aliased objects of subtype S, and components whose subtype is S, will have Size = 32 (assuming the implementation chooses to obey this Implementation Advice). On the other hand, if one writes, "**for** S2'Size **use** 5;" then stand-alone objects of subtype S2 will typically have their Size rounded up to ensure independent addressability.

52.b Note that "**for** S'Size **use** 32;" does not cause things like formal parameters to have Size = 32 — the implementation is allowed to make all parameters be at least 64 bits, for example.

52.c Note that "**for** S2'Size **use** 5;" requires record components whose subtype is S2 to be exactly 5 bits if the record type is packed. The same is not true of array components; their Size may be rounded up to the nearest factor of the word size.

52.d/2 **Implementation Note:** {*AI95-00291-02*} {*gaps*} On most machines, arrays don't contain gaps between elementary components; if the Component_Size is greater than the Size of the component subtype, the extra bits are generally considered part of each component, rather than gaps between components. On the other hand, a record might contain gaps between elementary components, depending on what sorts of loads, stores, and masking operations are generally done by the generated code.

52.e/2 {*AI95-00291-02*} For an array, any extra bits stored for each elementary component will generally be part of the component — the whole point of storing extra bits is to make loads and stores more efficient by avoiding the need to mask out extra bits. The PDP-10 is one counter-example; since the hardware supports byte strings with a gap at the end of each word, one would want to pack in that manner.

53 A Size clause on a composite subtype should not affect the internal layout of components.

53.a.1/2 **Implementation Advice:** A Size clause on a composite subtype should not affect the internal layout of components.

53.a **Reason:** That's what Pack pragmas, record_representation_clauses, and Component_Size clauses are for.

54 {*recommended level of support (Size attribute)* [partial]} The recommended level of support for the Size attribute of subtypes is:

- 55 The Size (if not specified) of a static discrete or fixed point subtype should be the number of bits needed to represent each value belonging to the subtype using an unbiased representation, leaving space for a sign bit only if the subtype contains negative values. If such a subtype is a first subtype, then an implementation should support a specified Size for it that reflects this representation.

55.a **Implementation Note:** This applies to static enumeration subtypes, using the internal codes used to represent the values.

55.b For a two's-complement machine, this implies that for a static signed integer subtype S, if all values of S are in the range 0 .. 2^n–1, or all values of S are in the range –2^{n-1} .. 2^{n-1}–1, for some *n* less than or equal to the word size, then S'Size should be <= the smallest such *n*. For a one's-complement machine, it is the same except that in the second range, the lower bound “–2^{n-1}” is replaced by “–2^{n-1}+1”.

55.c If an integer subtype (whether signed or unsigned) contains no negative values, the Size should not include space for a sign bit.

55.d Typically, the implementation will choose to make the Size of a subtype be exactly the smallest such *n*. However, it might, for example, choose a biased representation, in which case it could choose a smaller value.

55.e On most machines, it is in general not a good idea to pack (parts of) multiple stand-alone objects into the same storage element, because (1) it usually doesn't save much space, and (2) it requires locking to prevent tasks from interfering with each other, since separate stand-alone objects are independently addressable. Therefore, if S'Size = 2 on a machine with 8-bit storage elements, the size of a stand-alone object of subtype S will probably not be 2. It might, for example, be 8, 16 or 32, depending on the availability and efficiency of various machine instructions. The same applies to components of composite types, unless packing, Component_Size, or record layout is specified.

55.f For an unconstrained discriminated object, if the implementation allocates the maximum possible size, then the Size attribute should return that maximum possible size.

55.g **Ramification:** The Size of an object X is not usually the same as that of its subtype S. If X is a stand-alone object or a parameter, for example, most implementations will round X'Size up to a storage element boundary, or more, so X'Size might be greater than S'Size. On the other hand, X'Size cannot be less than S'Size, even if the implementation can prove, for example, that the range of values actually taken on by X during execution is smaller than the range of S.

55.h For example, if S is a first integer subtype whose range is 0..3, S'Size will be probably be 2 bits, and components of packed composite types of this subtype will be 2 bits (assuming Storage_Unit is a multiple of 2), but stand-alone objects and parameters will probably not have a size of 2 bits; they might be rounded up to 32 bits, for example. On the other hand, Unchecked_Conversion will use the 2-bit size, even when converting a stand-alone object, as one would expect.

55.i Another reason for making the Size of an object bigger than its subtype's Size is to support the run-time detection of uninitialized variables. {*uninitialized variables* [partial]} The implementation might add an extra value to a discrete subtype that represents the uninitialized state, and check for this value on use. In some cases, the extra value will require an extra bit in the representation of the object. Such detection is not required by the language. If it is provided, the implementation has to be able to turn it off. For example, if the programmer gives a record_representation_clause or Component_Size clause that makes a component too small to allow the extra bit, then the implementation will not be able to perform the checking (not using this method, anyway).

55.j The fact that the size of an object is not necessarily the same as its subtype can be confusing:

55.k
```
    type Device_Register is range 0..2**8 - 1;
    for Device_Register'Size use 8; -- Confusing!
    My_Device : Device_Register;
    for My_Device'Address use To_Address(16#FF00#);
```

55.l The programmer might think that My_Device'Size is 8, and that My_Device'Address points at an 8-bit location. However, this is not true. In Ada 83 (and in Ada 95), My_Device'Size might well be 32, and My_Device'Address might well point at the high-order 8 bits of the 32-bit object, which are always all zero bits. If My_Device'Address is passed to an assembly language subprogram, based on the programmer's assumption, the program will not work properly.

55.m **Reason:** It is not reasonable to require that an implementation allocate exactly 8 bits to all objects of subtype Device_Register. For example, in many run-time models, stand-alone objects and parameters are always aligned to a word boundary. Such run-time models are generally based on hardware considerations that are beyond the control of the implementer. (It is reasonable to require that an implementation allocate exactly 8 bits to all components of subtype Device_Register, if packed.)

55.n **Ramification:** The correct way to write the above code is like this:

55.o
```
    type Device_Register is range 0..2**8 - 1;
    My_Device : Device_Register;
    for My_Device'Size use 8;
    for My_Device'Address use To_Address(16#FF00#);
```

55.p If the implementation cannot accept 8-bit stand-alone objects, then this will be illegal. However, on a machine where an 8-bit device register exists, the implementation will probably be able to accept 8-bit stand-alone objects. Therefore, My_Device'Size will be 8, and My_Device'Address will point at those 8 bits, as desired.

55.q If an object of subtype Device_Register is passed to a foreign language subprogram, it will be passed according to that subprogram's conventions. Most foreign language implementations have similar run-time model restrictions. For example, when passing to a C function, where the argument is of the C type char* (that is, pointer to char), the C compiler will generally expect a full word value, either on the stack, or in a register. It will *not* expect a single byte. Thus, Size clauses for subtypes really have nothing to do with passing parameters to foreign language subprograms.

56 • For a subtype implemented with levels of indirection, the Size should include the size of the pointers, but not the size of what they point at.

56.a **Ramification:** For example, if a task object is represented as a pointer to some information (including a task stack), then the size of the object should be the size of the pointer. The Storage_Size, on the other hand, should include the size of the stack.

56.1/2 • {*AI95-00051-02*} An implementation should support a Size clause for a discrete type, fixed point type, record type, or array type, subject to the following:

56.2/2 • {*AI95-00051-02*} An implementation need not support a Size clause for a signed integer type specifying a Size greater than that of the largest signed integer type supported by the implementation in the absence of a size clause (that is, when the size is chosen by default). A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types.

56.b/2 **Discussion:** {*AI95-00051-02*} Note that the “corresponding limitation” for a record or array type implies that an implementation may impose some reasonable maximum size for records and arrays (e.g. 2**32 bits), which is an upper bound (“capacity” limit) on the size, whether chosen by default or by being specified by the user. The largest size supported for records need not be the same as the largest size supported for arrays.

56.3/2 • {*AI95-00291-02*} A nonconfirming size clause for the first subtype of a derived untagged by-reference type need not be supported.

56.c/2 **Implementation Advice:** The recommended level of support for the Size attribute should be followed.

56.d/2 **Ramification:** {*AI95-00291-02*} There is no recommendation to support any nonconfirming Size clauses for types not mentioned above. Remember that 13.1 requires support for confirming Size clauses for all types.

NOTES

57 5 Size is a subtype-specific attribute.

58 6 A component_clause or Component_Size clause can override a specified Size. A pragma Pack cannot.

Inconsistencies With Ada 83

{*AI95-00114-01*} We specify the meaning of Size in much more detail than Ada 83. This is not technically an inconsistency, but it is in practice, as most Ada 83 compilers use a different definition for Size than is required here. This should have been documented more explicitly during the Ada 9X process. 58.a.1/2

Wording Changes from Ada 83

The requirement for a nonnegative value in a Size clause was not in RM83, but it's hard to see how it would make sense. For uniformity, we forbid negative sizes, rather than letting implementations define their meaning. 58.a

Static Semantics

For a prefix T that denotes a task object [(after any implicit dereference)]: 59/1

T'Storage_Size 60

Denotes the number of storage elements reserved for the task. The value of this attribute is of the type *universal_integer*. The Storage_Size includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the "task control block" used by some implementations.) If a pragma Storage_Size is given, the value of the Storage_Size attribute is at least the value specified in the pragma.

Ramification: The value of this attribute is never negative, since it is impossible to "reserve" a negative number of storage elements. 60.a

If the implementation chooses to allocate an initial amount of storage, and then increase this as needed, the Storage_Size cannot include the additional amounts (assuming the allocation of the additional amounts can raise Storage_Error); this is inherent in the meaning of "reserved." 60.b

The implementation is allowed to allocate different amounts of storage for different tasks of the same subtype. 60.c

Storage_Size is also defined for access subtypes — see 13.11. 60.d

[{*Storage_Size clause: See also pragma Storage_Size*} A pragma Storage_Size specifies the amount of storage to be reserved for the execution of a task.] 61

Syntax

The form of a pragma Storage_Size is as follows: 62

pragma Storage_Size(expression); 63

A pragma Storage_Size is allowed only immediately within a task_definition. 64

Name Resolution Rules

{*expected type (Storage_Size pragma argument)* [partial]} The expression of a pragma Storage_Size is expected to be of any integer type. 65

Dynamic Semantics

A pragma Storage_Size is elaborated when an object of the type defined by the immediately enclosing task_definition is created. {*elaboration (Storage_Size pragma)* [partial]} For the elaboration of a pragma Storage_Size, the expression is evaluated; the Storage_Size attribute of the newly created task object is at least the value of the expression. 66

Ramification: The implementation is allowed to round up a specified Storage_Size amount. For example, if the implementation always allocates in chunks of 4096 bytes, the number 200 might be rounded up to 4096. Also, if the user specifies a negative number, the implementation has to normalize this to 0, or perhaps to a positive number. 66.a

{*Storage_Check* [partial]} {*check, language-defined (Storage_Check)*} {*Storage_Error (raised by failure of run-time check)*} At the point of task object creation, or upon task activation, Storage_Error is raised if there is insufficient free storage to accommodate the requested Storage_Size. 67

Static Semantics

68/1 For a prefix X that denotes an array subtype or array object [(after any implicit dereference)]:

69 X'Component_Size

Denotes the size in bits of components of the type of X. The value of this attribute is of type *universal_integer*.

70 {*specifiable (of Component_Size for array types)* [partial]} {*Component_Size clause*} Component_Size may be specified for array types via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative.

70.a **Implementation Note:** The intent is that the value of X'Component_Size is always nonnegative. If the array is stored “backwards” in memory (which might be caused by an implementation-defined pragma), X'Component_Size is still positive.

70.b **Ramification:** For an array object A, A'Component_Size = A(I)'Size for any index I.

Implementation Advice

71 {*recommended level of support (Component_Size attribute)* [partial]} The recommended level of support for the Component_Size attribute is:

72 • An implementation need not support specified Component_Sizes that are less than the Size of the component subtype.

73 • An implementation should support specified Component_Sizes that are factors and multiples of the word size. For such Component_Sizes, the array should contain no gaps between components. For other Component_Sizes (if supported), the array should contain no gaps between components when packing is also specified; the implementation should forbid this combination in cases where it cannot support a no-gaps representation.

73.a **Ramification:** For example, if Storage_Unit = 8, and Word_Size = 32, then the user is allowed to specify a Component_Size of 1, 2, 4, 8, 16, and 32, with no gaps. In addition, *n**32 is allowed for positive integers *n*, again with no gaps. If the implementation accepts Component_Size = 3, then it might allocate 10 components per word, with a 2-bit gap at the end of each word (unless packing is also specified), or it might not have any internal gaps at all. (There can be gaps at either end of the array.)

73.b/2 **Implementation Advice:** The recommended level of support for the Component_Size attribute should be followed.

Static Semantics

73.1/1 {*8652/0009*} {*AI95-00137-01*} The following operational attribute is defined: External_Tag.

74/1 {*8652/0009*} {*AI95-00137-01*} For every subtype S of a tagged type *T* (specific or class-wide):

75/1 S'External_Tag

{*8652/0040*} {*AI95-00108-01*} {*External_Tag clause*} {*specifiable (of External_Tag for a tagged type)* [partial]} S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined. See 3.9.2 and 13.13.2. The value of External_Tag is never inherited[; the default value is always used unless a new value is directly specified for a type].

75.a **Implementation defined:** The default external representation for a type tag.

Implementation Requirements

76 In an implementation, the default external tag for each specific tagged type declared in a partition shall be distinct, so long as the type is declared outside an instance of a generic body. If the compilation unit in which a given tagged type is declared, and all compilation units on which it semantically depends, are the same in two different partitions, then the external tag for the type shall be the same in the two partitions. What it means for a compilation unit to be the same in two different partitions is implementation defined.

At a minimum, if the compilation unit is not recompiled between building the two different partitions that include it, the compilation unit is considered the same in the two partitions.

76.a **Implementation defined:** What determines whether a compilation unit is the same in two different partitions.

76.b **Reason:** These requirements are important because external tags are used for input/output of class-wide types. These requirements ensure that what is written by one program can be read back by some other program so long as they share the same declaration for the type (and everything it depends on).

76.c The user may specify the external tag if (s)he wishes its value to be stable even across changes to the compilation unit in which the type is declared (or changes in some unit on which it depends).

76.d/2 {*AI95-00114-01*} We use a String rather than a Stream_Element_Array to represent an external tag for portability.

76.e **Ramification:** Note that the characters of an external tag need not all be graphic characters. In other words, the external tag can be a sequence of arbitrary 8-bit bytes.

NOTES

77/2 7 {*AI95-00270-01*} The following language-defined attributes are specifiable, at least for some of the kinds of entities to which they apply: Address, Alignment, Bit_Order, Component_Size, External_Tag, Input, Machine_Radix, Output, Read, Size, Small, Storage_Pool, Storage_Size, Stream_Size, and Write.

78 8 It follows from the general rules in 13.1 that if one writes “**for** X'Size **use** Y;” then the X'Size attribute_reference will return Y (assuming the implementation allows the Size clause). The same is true for all of the specifiable attributes except Storage_Size.

78.a **Ramification:** An implementation may specify that an implementation-defined attribute is specifiable for certain entities. This follows from the fact that the semantics of implementation-defined attributes is implementation defined. An implementation is not allowed to make a language-defined attribute specifiable if it isn't.

Examples

79 *Examples of attribute definition clauses:*

80
```
Byte : constant := 8;
Page : constant := 2**12;
```

81
```
type Medium is range 0 .. 65_000;
for Medium'Size use 2*Byte;
for Medium'Alignment use 2;
Device_Register : Medium;
for Device_Register'Size use Medium'Size;
for Device_Register'Address use System.Storage_Elements.To_Address(16#FFFF_0020#);
```

82
```
type Short is delta 0.01 range -100.0 .. 100.0;
for Short'Size use 15;
```

83
```
for Car_Name'Storage_Size use -- specify access type's storage pool size
        2000*((Car'Size/System.Storage_Unit) +1); -- approximately 2000 cars
```

84/2
```
{AI95-00441-01} function My_Input(Stream : not null access
Ada.Streams.Root_Stream_Type'Class)
  return T;
for T'Input use My_Input; -- see 13.13.2
```

NOTES

85 9 *Notes on the examples:* In the Size clause for Short, fifteen bits is the minimum necessary, since the type definition requires Short'Small <= 2**(–7).

Extensions to Ada 83

85.a {*extensions to Ada 83*} The syntax rule for length_clause is replaced with the new syntax rule for attribute_definition_clause, and it is modified to allow a name (as well as an expression).

Wording Changes from Ada 83

85.b The syntax rule for attribute_definition_clause now requires that the prefix of the attribute be a local_name; in Ada 83 this rule was stated in the text.

85.c/2 {*AI95-00114-01*} In Ada 83, the relationship between a aspect_clause specifying a certain aspect and an attribute that queried that aspect was unclear. In Ada 95, they are the same, except for certain explicit exceptions.

Wording Changes from Ada 95

85.d/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added wording to specify for each attribute whether it is an operational or representation attribute.

85.e/2 {*8652/0040*} {*AI95-00108-01*} **Corrigendum:** Added wording to specify that External_Tag is never inherited.

85.f/2 {*AI95-00051-01*} {*AI95-00291-01*} Adjusted the Recommended Level of Support for Alignment to eliminate nonsense requirements and to ensure that useful capabilities are required.

85.g/2 {*AI95-00051-01*} {*AI95-00291-01*} Adjusted the Recommended Level of Support for Size to eliminate nonsense requirements and to ensure that useful capabilities are required. Also eliminated any dependence on whether an aspect was specified (a confirming representation item should not affect the semantics).

85.h/2 {*AI95-00133-01*} Added the definition of machine scalar.

85.i/2 {*AI95-00247-01*} Removed the requirement that specified alignments for a composite type cannot override those for their components, because it was never intended to apply to components whose location was specified with a representation item. Moreover, it causes a difference in legality when a confirming alignment is specified for one of the composite types.

85.j/2 {*AI95-00291-02*} Removed recommended level of support rules about types with by-reference and aliased parts, because there are now blanket rules covering all recommended level of support rules.

85.k/2 {*AI95-00291-02*} Split the definition of Alignment for subtypes and for objects. This simplified the wording and eliminated confusion about which rules applied to objects, which applied to subtypes, and which applied to both.

13.4 Enumeration Representation Clauses

1 [An enumeration_representation_clause specifies the internal codes for enumeration literals.]

Syntax

2 enumeration_representation_clause ::=
  for *first_subtype*_local_name **use** enumeration_aggregate;

3 enumeration_aggregate ::= array_aggregate

Name Resolution Rules

4 {*expected type (enumeration_representation_clause expressions)* [partial]} The enumeration_aggregate shall be written as a one-dimensional array_aggregate, for which the index subtype is the unconstrained subtype of the enumeration type, and each component expression is expected to be of any integer type.

4.a **Ramification:** The “full coverage rules” for aggregates applies. An **others** is not allowed — there is no applicable index constraint in this context.

Legality Rules

5 The *first_subtype*_local_name of an enumeration_representation_clause shall denote an enumeration subtype.

5.a **Ramification:** As for all type-related representation items, the local_name is required to denote a first subtype.

6/2 {*AI95-00287-01*} Each component of the array_aggregate shall be given by an expression rather than a <>. The expressions given in the array_aggregate shall be static, and shall specify distinct integer codes for each value of the enumeration type; the associated integer codes shall satisfy the predefined ordering relation of the type.

6.a **Reason:** Each value of the enumeration type has to be given an internal code, even if the first subtype of the enumeration type is constrained to only a subrange (this is only possible if the enumeration type is a derived type). This “full coverage” requirement is important because one may refer to Enum'Base'First and Enum'Base'Last, which need to have defined representations.

Static Semantics

{*aspect of representation (coding)* [partial]} {*coding (aspect of representation)*} An enumeration_representation_clause specifies the *coding* aspect of representation. {*internal code*} The coding consists of the *internal code* for each enumeration literal, that is, the integral value used internally to represent each literal. 7

Implementation Requirements

For nonboolean enumeration types, if the coding is not specified for the type, then for each value of the type, the internal code shall be equal to its position number. 8

Reason: This default representation is already used by all known Ada compilers for nonboolean enumeration types. Therefore, we make it a requirement so users can depend on it, rather than feeling obliged to supply for every enumeration type an enumeration representation clause that is equivalent to this default rule. 8.a

Discussion: For boolean types, it is relatively common to use all ones for True, and all zeros for False, since some hardware supports that directly. Of course, for a one-bit Boolean object (like in a packed array), False is presumably zero and True is presumably one (choosing the reverse would be extremely unfriendly!). 8.b

Implementation Advice

{*recommended level of support (enumeration_representation_clause)* [partial]} The recommended level of support for enumeration_representation_clauses is: 9

- An implementation should support at least the internal codes in the range System.Min_Int..System.Max_Int. An implementation need not support enumeration_-representation_clauses for boolean types. 10

Ramification: The implementation may support numbers outside the above range, such as numbers greater than System.Max_Int. See AI83-00564. 10.a

Reason: The benefits of specifying the internal coding of a boolean type do not outweigh the implementation costs. Consider, for example, the implementation of the logical operators on a packed array of booleans with strange internal codes. It's implementable, but not worth it. 10.b

Implementation Advice: The recommended level of support for enumeration_representation_clauses should be followed. 10.c/2

NOTES

10 {*8652/0009*} {*AI95-00137-01*} Unchecked_Conversion may be used to query the internal codes used for an enumeration type. The attributes of the type, such as Succ, Pred, and Pos, are unaffected by the enumeration_representation_clause. For example, Pos always returns the position number, *not* the internal integer code that might have been specified in an enumeration_representation_clause}. 11/1

Discussion: Suppose the enumeration type in question is derived: 11.a

11.b
```
type T1 is (Red, Green, Blue);
subtype S1 is T1 range Red .. Green;
type S2 is new S1;
for S2 use (Red => 10, Green => 20, Blue => 30);
```

{*8652/0009*} {*AI95-00137-01*} The enumeration_representation_clause has to specify values for all enumerals, even ones that are not in S2 (such as Blue). The Base attribute can be used to get at these values. For example: 11.c/1

11.d
```
for I in S2'Base loop
    ... -- When I equals Blue, the internal code is 30.
end loop;
```

We considered allowing or requiring “**for** S2'Base **use** ...” in cases like this, but it didn't seem worth the trouble. 11.e

Examples

Example of an enumeration representation clause: 12

13
```
type Mix_Code is (ADD, SUB, MUL, LDA, STA, STZ);
```

14
```
for Mix_Code use
   (ADD => 1, SUB => 2, MUL => 3, LDA => 8, STA => 24, STZ =>33);
```

Extensions to Ada 83

14.a {*extensions to Ada 83*} As in other similar contexts, Ada 95 allows expressions of any integer type, not just expressions of type *universal_integer*, for the component expressions in the enumeration_aggregate. The preference rules for the predefined operators of *root_integer* eliminate any ambiguity.

14.b For portability, we now require that the default coding for an enumeration type be the "obvious" coding using position numbers. This is satisfied by all known implementations.

Wording Changes from Ada 95

14.c/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Updated to reflect that we no longer have something called representation_clause.

14.d/2 {*AI95-00287-01*} Added wording to prevent the use of <> in a enumeration_representation_clause. (<> is newly added to array_aggregates.)

13.5 Record Layout

1 {*aspect of representation (layout)* [partial]} {*layout (aspect of representation)*} {*aspect of representation (record layout)* [partial]} {*record layout (aspect of representation)*} {*aspect of representation (storage place)* [partial]} {*storage place (of a component)*} The *(record) layout* aspect of representation consists of the *storage places* for some or all components, that is, storage place attributes of the components. The layout can be specified with a record_representation_clause.

13.5.1 Record Representation Clauses

1 [A record_representation_clause specifies the storage representation of records and record extensions, that is, the order, position, and size of components (including discriminants, if any). {*bit field: See record_representation_clause*}]

Language Design Principles

1.a/2 {*AI95-00114-01*} It should be feasible for an implementation to use negative offsets in the representation of composite types. However, no implementation should be forced to support negative offsets. Therefore, in the interest of uniformity, negative offsets should be disallowed in record_representation_clauses.

Syntax

2
```
record_representation_clause ::=
   for first_subtype_local_name use
    record [mod_clause]
      {component_clause}
    end record;
```

3
```
component_clause ::=
   component_local_name at position range first_bit .. last_bit;
```

4 position ::= *static*_expression

5 first_bit ::= *static*_simple_expression

6 last_bit ::= *static*_simple_expression

6.a **Reason:** First_bit and last_bit need to be simple_expression instead of expression for the same reason as in range (see 3.5, "Scalar Types").

Name Resolution Rules

7 {*expected type (component_clause expressions)* [partial]} {*expected type (position)* [partial]} {*expected type (first_bit)* [partial]} {*expected type (last_bit)* [partial]} Each position, first_bit, and last_bit is expected to be of any integer type.

Ramification: These need not have the same integer type. 7.a

Legality Rules

{*AI95-00436-01*} The *first_subtype*_local_name of a record_representation_clause shall denote a specific record or record extension subtype. 8/2

Ramification: As for all type-related representation items, the local_name is required to denote a first subtype. 8.a

If the *component*_local_name is a direct_name, the local_name shall denote a component of the type. For a record extension, the component shall not be inherited, and shall not be a discriminant that corresponds to a discriminant of the parent type. If the *component*_local_name has an attribute_designator, the direct_name of the local_name shall denote either the declaration of the type or a component of the type, and the attribute_designator shall denote an implementation-defined implicit component of the type. 9

The position, first_bit, and last_bit shall be static expressions. The value of position and first_bit shall be nonnegative. The value of last_bit shall be no less than first_bit – 1. 10

Ramification: A component_clause such as "X **at** 4 **range** 0..–1;" is allowed if X can fit in zero bits. 10.a

{*AI95-00133-01*} If the nondefault bit ordering applies to the type, then either: 10.1/2

- the value of last_bit shall be less than the size of the largest machine scalar; or 10.2/2
- the value of first_bit shall be zero and the value of last_bit + 1 shall be a multiple of System.Storage_Unit. 10.3/2

At most one component_clause is allowed for each component of the type, including for each discriminant (component_clauses may be given for some, all, or none of the components). Storage places within a component_list shall not overlap, unless they are for components in distinct variants of the same variant_part. 11

A name that denotes a component of a type is not allowed within a record_representation_clause for the type, except as the *component*_local_name of a component_clause. 12

Reason: It might seem strange to make the record_representation_clause part of the declarative region, and then disallow mentions of the components within almost all of the record_representation_clause. The alternative would be to treat the *component*_local_name like a formal parameter name in a subprogram call (in terms of visibility). However, this rule would imply slightly different semantics, because (given the actual rule) the components can hide other declarations. This was the rule in Ada 83, and we see no reason to change it. The following, for example, was and is illegal: 12.a

12.b
```
type T is
    record
        X : Integer;
    end record;
X : constant := 31; -- Same defining name as the component.
for T use
    record
        X at 0 range 0..X; -- Illegal!
    end record;
```

The component X hides the named number X throughout the record_representation_clause. 12.c

Static Semantics

{*AI95-00133-01*} A record_representation_clause (without the mod_clause) specifies the layout. 13/2

{*AI95-00133-01*} If the default bit ordering applies to the type, the position, first_bit, and last_bit of each component_clause directly specify the position and size of the corresponding component. 13.1/2

{*AI95-00133-01*} If the nondefault bit ordering applies to the type then the layout is determined as follows: 13.2/2

13.3/2 • the component_clauses for which the value of last_bit is greater than or equal to the size of the largest machine scalar directly specify the position and size of the corresponding component;

13.4/2 • for other component_clauses, all of the components having the same value of position are considered to be part of a single machine scalar, located at that position; this machine scalar has a size which is the smallest machine scalar size larger than the largest last_bit for all component_clauses at that position; the first_bit and last_bit of each component_clause are then interpreted as bit offsets in this machine scalar.

13.a/2 *This paragraph was deleted.*{*AI95-00133-01*}

13.b **Ramification:** A component_clause also determines the value of the Size attribute of the component, since this attribute is related to First_Bit and Last_Bit.

14 [A record_representation_clause for a record extension does not override the layout of the parent part;] if the layout was specified for the parent type, it is inherited by the record extension.

Implementation Permissions

15 An implementation may generate implementation-defined components (for example, one containing the offset of another component). An implementation may generate names that denote such implementation-defined components; such names shall be implementation-defined attribute_references. An implementation may allow such implementation-defined names to be used in record_representation_clauses. An implementation can restrict such component_clauses in any manner it sees fit.

15.a **Implementation defined:** Implementation-defined components.

15.b **Ramification:** Of course, since the semantics of implementation-defined attributes is implementation defined, the implementation need not support these names in all situations. They might be purely for the purpose of component_clauses, for example. The visibility rules for such names are up to the implementation.

15.c We do not allow such component names to be normal identifiers — that would constitute blanket permission to do all kinds of evil things.

15.d **Discussion:** {*dope*} Such implementation-defined components are known in the vernacular as “dope.” Their main purpose is for storing offsets of components that depend on discriminants.

16 If a record_representation_clause is given for an untagged derived type, the storage place attributes for all of the components of the derived type may differ from those of the corresponding components of the parent type, even for components whose storage place is not specified explicitly in the record_-representation_clause.

16.a **Reason:** This is clearly necessary, since the whole record may need to be laid out differently.

Implementation Advice

17 {*recommended level of support (record_representation_clause)* [partial]} The recommended level of support for record_representation_clauses is:

17.1/2 • {*AI95-00133-01*} An implementation should support machine scalars that correspond to all of the integer, floating point, and address formats supported by the machine.

18 • An implementation should support storage places that can be extracted with a load, mask, shift sequence of machine code, and set with a load, shift, mask, store sequence, given the available machine instructions and run-time model.

19 • A storage place should be supported if its size is equal to the Size of the component subtype, and it starts and ends on a boundary that obeys the Alignment of the component subtype.

20/2 • {*AI95-00133-01*} For a component with a subtype whose Size is less than the word size, any storage place that does not cross an aligned word boundary should be supported.

20.a **Reason:** The above recommendations are sufficient to define interfaces to most interesting hardware. This causes less implementation burden than the definition in ACID, which requires arbitrary bit alignments of arbitrarily large components. Since the ACID definition is neither enforced by the ACVC, nor supported by all implementations, it seems OK for us to weaken it.

- An implementation may reserve a storage place for the tag field of a tagged type, and disallow other components from overlapping that place. 21

Ramification: Similar permission for other dope is not granted. 21.a

- An implementation need not support a component_clause for a component of an extension part if the storage place is not after the storage places of all components of the parent type, whether or not those storage places had been specified. 22

Reason: These restrictions are probably necessary if block equality operations are to be feasible for class-wide types. For block comparison to work, the implementation typically has to fill in any gaps with zero (or one) bits. If a "gap" in the parent type is filled in with a component in a type extension, then this won't work when a class-wide object is passed by reference, as is required. 22.a

Implementation Advice: The recommended level of support for record_representation_clauses should be followed. 22.b/2

NOTES

11 If no component_clause is given for a component, then the choice of the storage place for the component is left to the implementation. If component_clauses are given for all components, the record_representation_clause completely specifies the representation of the type and will be obeyed exactly by the implementation. 23

Ramification: The visibility rules prevent the name of a component of the type from appearing in a record_representation_clause at any place *except* for the *component_*local_name of a component_clause. However, since the record_representation_clause is part of the declarative region of the type declaration, the component names hide outer homographs throughout. 23.a

{*8652/0009*} {*AI95-00137-01*} A record_representation_clause cannot be given for a protected type, even though protected types, like record types, have components. The primary reason for this rule is that there is likely to be too much dope in a protected type — entry queues, bit maps for barrier values, etc. In order to control the representation of the user-defined components, simply declare a record type, give it a record_representation_clause, and give the protected type one component whose type is the record type. Alternatively, if the protected object is protecting something like a device register, it makes more sense to keep the thing being protected outside the protected object (possibly with a pointer to it in the protected object), in order to keep implementation-defined components out of the way. 23.b/1

Examples

Example of specifying the layout of a record type: 24

25
```
Word : constant := 4;  --  storage element is byte, 4 bytes per word
```

26
```
type State         is (A,M,W,P);
type Mode          is (Fix, Dec, Exp, Signif);
```

27
```
type Byte_Mask     is array (0..7)  of Boolean;
type State_Mask    is array (State) of Boolean;
type Mode_Mask     is array (Mode)  of Boolean;
```

28
```
type Program_Status_Word is
  record
      System_Mask        : Byte_Mask;
      Protection_Key     : Integer range 0 .. 3;
      Machine_State      : State_Mask;
      Interrupt_Cause    : Interruption_Code;
      Ilc                : Integer range 0 .. 3;
      Cc                 : Integer range 0 .. 3;
      Program_Mask       : Mode_Mask;
      Inst_Address       : Address;
end record;
```

29
```
for Program_Status_Word use
  record
      System_Mask      at 0*Word range 0  .. 7;
      Protection_Key   at 0*Word range 10 .. 11; -- bits 8,9 unused
      Machine_State    at 0*Word range 12 .. 15;
      Interrupt_Cause  at 0*Word range 16 .. 31;
      Ilc              at 1*Word range 0  .. 1;  -- second word
      Cc               at 1*Word range 2  .. 3;
      Program_Mask     at 1*Word range 4  .. 7;
      Inst_Address     at 1*Word range 8  .. 31;
  end record;
```

30
```
for Program_Status_Word'Size use 8*System.Storage_Unit;
for Program_Status_Word'Alignment use 8;
```

NOTES

31 12 *Note on the example:* The record_representation_clause defines the record layout. The Size clause guarantees that (at least) eight storage elements are used for objects of the type. The Alignment clause guarantees that aliased, imported, or exported objects of the type will have addresses divisible by eight.

Wording Changes from Ada 83

31.a The alignment_clause has been renamed to mod_clause and moved to Annex J, “Obsolescent Features”.

31.b We have clarified that implementation-defined component names have to be in the form of an attribute_reference of a component or of the first subtype itself; surely Ada 83 did not intend to allow arbitrary identifiers.

31.c The RM83-13.4(7) wording incorrectly allows components in non-variant records to overlap. We have corrected that oversight.

Incompatibilities With Ada 95

31.d/2 {*AI95-00133-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** The meaning of a record_representation_clause for the nondefault bit order is now clearly defined. Thus, such clauses can be portably written. In order to do that though, the equivalence of bit 1 in word 1 to bit 9 in word 0 (for a machine with Storage_Unit = 8) had to be dropped for the nondefault bit order. Any record_representation_clauses which depends on that equivalence will break (although such code would imply a non-contiguous representation for a component, and it seems unlikely that compilers were supporting that anyway).

Extensions to Ada 95

31.e/2 {*AI95-00436-01*} {*extensions to Ada 95*} **Amendment Correction:** The undocumented (and likely unintentional) incompatibility with Ada 83 caused by not allowing record_representation_clauses on limited record types is removed.

13.5.2 Storage Place Attributes

Static Semantics

1 {*storage place attributes (of a component)*} For a component C of a composite, non-array object R, the *storage place attributes* are defined:

1.a **Ramification:** The storage place attributes are not (individually) specifiable, but the user may control their values by giving a record_representation_clause.

2/2 R.C'Position

{*AI95-00133-01*} If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the position of the component_clause; otherwise, denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*.

2.a/2 **Ramification:** {*AI95-00133-01*} Thus, for the default bit order, R.C'Position is the offset of C in storage elements from the beginning of the object, where the first storage element of an object is numbered zero. R'Address + R.C'Position = R.C'Address. For record extensions, the offset is not measured from the beginning of the extension part, but from the beginning of the whole object, as usual.

2.b In “R.C'Address – R'Address”, the "–" operator is the one in System.Storage_Elements that takes two Addresses and returns a Storage_Offset.

R.C'First_Bit 3/2

{*AI95-00133-01*} If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the first_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*.

R.C'Last_Bit 4/2

{*AI95-00133-01*} If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the last_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer*.

Ramification: {*AI95-00114-01*} The ordering of bits in a storage element is defined in 13.5.3, "Bit Ordering". 4.a/2

R.C'Size = R.C'Last_Bit – R.C'First_Bit + 1. (Unless the implementation chooses an indirection representation.) 4.b

If a component_clause applies to a component, then that component will be at the same relative storage place in all objects of the type. Otherwise, there is no such requirement. 4.c

Implementation Advice

{*contiguous representation* [partial]} {*discontiguous representation* [partial]} If a component is represented using some form of pointer (such as an offset) to the actual data of the component, and this data is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data, not the pointer. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes. 5

Reason: For discontiguous components, these attributes make no sense. For example, an implementation might allocate dynamic-sized components on the heap. For another example, an implementation might allocate the discriminants separately from the other components, so that multiple objects of the same subtype can share discriminants. Such representations cannot happen if there is a component_clause for that component. 5.a

Implementation Advice: If a component is represented using a pointer to the actual data of the component which is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes. 5.b/2

Incompatibilities With Ada 95

{*AI95-00133-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** The meaning of the storage place attributes for the nondefault bit order is now clearly defined, and can be different than that given by strictly following the Ada 95 wording. Any code which depends on the original Ada 95 values for a type using the nondefault bit order where they are different will break. 5.c/2

13.5.3 Bit Ordering

[The Bit_Order attribute specifies the interpretation of the storage place attributes.] 1

Reason: The intention is to provide uniformity in the interpretation of storage places across implementations on a particular machine by allowing the user to specify the Bit_Order. It is not intended to fully support data interoperability across different machines, although it can be used for that purpose in some situations. 1.a

{*AI95-00114-01*} We can't require all implementations on a given machine to use the same bit ordering by default; if the user cares, a Bit_Order attribute_definition_clause can be used to force all implementations to use the same bit ordering. 1.b/2

Static Semantics

{*bit ordering*} A bit ordering is a method of interpreting the meaning of the storage place attributes. {*High_Order_First*} {*big endian*} {*endian (big)*} High_Order_First [(known in the vernacular as "big 2 endian”)] means that the first bit of a storage element (bit 0) is the most significant bit (interpreting the sequence of bits that represent a component as an unsigned integer value). {*Low_Order_First*} {*little endian*} {*endian (little)*} Low_Order_First [(known in the vernacular as “little endian”)] means the opposite: the first bit is the least significant.

3 For every specific record subtype S, the following attribute is defined:

4 S'Bit_Order

Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. {*specifiable (of Bit_Order for record types and record extensions)* [partial]} {*Bit_Order clause*} Bit_Order may be specified for specific record types via an attribute_definition_clause; the expression of such a clause shall be static.

5 If Word_Size = Storage_Unit, the default bit ordering is implementation defined. If Word_Size > Storage_Unit, the default bit ordering is the same as the ordering of storage elements in a word, when interpreted as an integer. {*byte sex: See ordering of storage elements in a word*}

5.a **Implementation defined:** If Word_Size = Storage_Unit, the default bit ordering.

5.b **Ramification:** Consider machines whose Word_Size = 32, and whose Storage_Unit = 8. Assume the default bit ordering applies. On a machine with big-endian addresses, the most significant storage element of an integer is at the address of the integer. Therefore, bit zero of a storage element is the most significant bit. On a machine with little-endian addresses, the least significant storage element of an integer is at the address of the integer. Therefore, bit zero of a storage element is the least significant bit.

6 The storage place attributes of a component of a type are interpreted according to the bit ordering of the type.

6.a **Ramification:** This implies that the interpretation of the position, first_bit, and last_bit of a component_clause of a record_representation_clause obey the bit ordering given in a representation item.

Implementation Advice

7 {*recommended level of support (bit ordering)* [partial]} The recommended level of support for the nondefault bit ordering is:

8/2 • {*AI95-00133-01*} The implementation should support the nondefault bit ordering in addition to the default bit ordering.

8.a/2 **Ramification:** {*AI95-00133-01*} The implementation should support both bit orderings. Implementations are required to support storage positions that cross storage element boundaries when Word_Size > Storage_Unit but the definition of the storage place attributes for the nondefault bit order ensures that such storage positions will not be split into two or three pieces. Thus, there is no significant implementation burden to supporting the nondefault bit order, given that the set of machine scalars is implementation-defined.

8.b/2 **Implementation Advice:** The recommended level of support for the nondefault bit ordering should be followed.

NOTES

9/2 13 {*AI95-00133-01*} Bit_Order clauses make it possible to write record_representation_clauses that can be ported between machines having different bit ordering. They do not guarantee transparent exchange of data between such machines.

Extensions to Ada 83

9.a {*extensions to Ada 83*} The Bit_Order attribute is new to Ada 95.

Wording Changes from Ada 95

9.b/2 {*AI95-00133-01*} We now suggest that all implementations support the nondefault bit order.

13.6 Change of Representation

1 [{*change of representation*} {*representation (change of)*} A type_conversion (see 4.6) can be used to convert between two different representations of the same array or record. To convert an array from one

representation to another, two array types need to be declared with matching component subtypes, and convertible index types. If one type has packing specified and the other does not, then explicit conversion can be used to pack or unpack an array.

To convert a record from one representation to another, two record types with a common ancestor type need to be declared, with no inherited subprograms. Distinct representations can then be specified for the record types, and explicit conversion between the types can be used to effect a change in representation.] 2

Ramification: This technique does not work if the first type is an untagged type with user-defined primitive subprograms. It does not work at all for tagged types. 2.a

Examples

Example of change of representation: 3

```
-- Packed_Descriptor and Descriptor are two different types          4
-- with identical characteristics, apart from their
-- representation

type Descriptor is                                                   5
    record
      -- components of a descriptor
    end record;

type Packed_Descriptor is new Descriptor;                            6

for Packed_Descriptor use                                            7
    record
      -- component clauses for some or for all components
    end record;

-- Change of representation can now be accomplished by explicit type conversions:   8

D : Descriptor;                                                      9
P : Packed_Descriptor;

P := Packed_Descriptor(D);  -- pack D                                10
D := Descriptor(P);         -- unpack P
```

13.7 The Package System

[For each implementation there is a library package called System which includes the definitions of certain configuration-dependent characteristics.] 1

Static Semantics

The following language-defined library package exists: 2

Implementation defined: The contents of the visible part of package System. 2.a/2

```
{AI95-00362-01} package System is                                     3/2
   pragma Pure(System);

   type Name is implementation-defined-enumeration-type;              4
   System_Name : constant Name := implementation-defined;

   -- System-Dependent Named Numbers:                                 5

   Min_Int               : constant := root_integer'First;            6
   Max_Int               : constant := root_integer'Last;

   Max_Binary_Modulus    : constant := implementation-defined;        7
   Max_Nonbinary_Modulus : constant := implementation-defined;

   Max_Base_Digits       : constant := root_real'Digits;              8
   Max_Digits            : constant := implementation-defined;

   Max_Mantissa          : constant := implementation-defined;        9
   Fine_Delta            : constant := implementation-defined;

   Tick                  : constant := implementation-defined;        10
```

```
11        -- Storage-related Declarations:

12        type Address is implementation-defined;
          Null_Address : constant Address;

13        Storage_Unit : constant := implementation-defined;
          Word_Size    : constant := implementation-defined * Storage_Unit;
          Memory_Size  : constant := implementation-defined;

14        -- {address (comparison)} Address Comparison:
          function "<" (Left, Right : Address) return Boolean;
          function "<="(Left, Right : Address) return Boolean;
          function ">" (Left, Right : Address) return Boolean;
          function ">="(Left, Right : Address) return Boolean;
          function "=" (Left, Right : Address) return Boolean;
       -- function "/=" (Left, Right : Address) return Boolean;
          -- "/=" is implicitly defined
          pragma Convention(Intrinsic, "<");
          ... -- and so on for all language-defined subprograms in this package

15/2   {AI95-00221-01}     -- Other System-Dependent Declarations:
          type Bit_Order is (High_Order_First, Low_Order_First);
          Default_Bit_Order : constant Bit_Order := implementation-defined;

16        -- Priority-related declarations (see D.1):
          subtype Any_Priority is Integer range implementation-defined;
          subtype Priority is Any_Priority range Any_Priority'First ..
                    implementation-defined;
          subtype Interrupt_Priority is Any_Priority range Priority'Last+1 ..
                    Any_Priority'Last;

17        Default_Priority : constant Priority :=
                    (Priority'First + Priority'Last)/2;

18     private
          ... -- not specified by the language
       end System;
```

19 Name is an enumeration subtype. Values of type Name are the names of alternative machine configurations handled by the implementation. System_Name represents the current machine configuration.

20 The named numbers Fine_Delta and Tick are of the type *universal_real*; the others are of the type *universal_integer*.

21 The meanings of the named numbers are:

22 [Min_Int The smallest (most negative) value allowed for the expressions of a signed_integer_type_definition.

23 Max_Int The largest (most positive) value allowed for the expressions of a signed_integer_type_definition.

24 Max_Binary_Modulus
A power of two such that it, and all lesser positive powers of two, are allowed as the modulus of a modular_type_definition.

25 Max_Nonbinary_Modulus
A value such that it, and all lesser positive integers, are allowed as the modulus of a modular_type_definition.

25.a **Ramification:** There is no requirement that Max_Nonbinary_Modulus be less than or equal to Max_Binary_Modulus, although that's what makes most sense. On a typical 32-bit machine, for example, Max_Binary_Modulus will be 2**32 and Max_Nonbinary_Modulus will be 2**31, because supporting nonbinary moduli in above 2**31 causes implementation difficulties.

26 Max_Base_Digits
The largest value allowed for the requested decimal precision in a floating_point_definition.

Max_Digits 27
The largest value allowed for the requested decimal precision in a floating_point_definition that has no real_range_specification. Max_Digits is less than or equal to Max_Base_Digits.

Max_Mantissa 28
The largest possible number of binary digits in the mantissa of machine numbers of a user-defined ordinary fixed point type. (The mantissa is defined in Annex G.)

Fine_Delta 29
The smallest delta allowed in an ordinary_fixed_point_definition that has the real_range_-specification **range** –1.0 .. 1.0.]

Tick A period in seconds approximating the real time interval during which the value of Calendar.Clock remains constant. 30

Ramification: There is no required relationship between System.Tick and Duration'Small, other than the one described here. 30.a

The inaccuracy of the delay_statement has no relation to Tick. In particular, it is possible that the clock used for the delay_statement is less accurate than Calendar.Clock. 30.b

We considered making Tick a run-time-determined quantity, to allow for easier configurability. However, this would not be upward compatible, and the desired configurability can be achieved using functionality defined in Annex D, “Real-Time Systems”. 30.c

Storage_Unit 31
The number of bits per storage element.

Word_Size 32
The number of bits per word.

Memory_Size 33
An implementation-defined value [that is intended to reflect the memory size of the configuration in storage elements.]

Discussion: It is unspecified whether this refers to the size of the address space, the amount of physical memory on the machine, or perhaps some other interpretation of “memory size.” In any case, the value has to be given by a static expression, even though the amount of memory on many modern machines is a dynamic quantity in several ways. Thus, Memory_Size is not very useful. 33.a

{*AI95-00161-01*} Address is a definite, nonlimited type with preelaborable initialization (see 10.2.1). Address represents machine addresses capable of addressing individual storage elements. Null_Address is an address that is distinct from the address of any object or program unit. {*pointer: See type System.Address*} 34/2

Ramification: The implementation has to ensure that there is at least one address that nothing will be allocated to; Null_Address will be one such address. 34.a

Ramification: Address is the type of the result of the attribute Address. 34.b

Reason: Address is required to be nonlimited and definite because it is important to be able to assign addresses, and to declare uninitialized address variables. 34.c

Ramification: {*AI95-00161-01*} If System.Address is defined as a private type (as suggested below), it might be necessary to add a pragma Preelaborable_Initialization to the specification of System in order that Address have preelaborable initialization as required. 34.d/2

{*AI95-00221-01*} Default_Bit_Order shall be a static constant. See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order. 35/2

Implementation Permissions

{*AI95-00362-01*} An implementation may add additional implementation-defined declarations to package System and its children. [However, it is usually better for the implementation to provide additional functionality via implementation-defined children of System.] 36/2

36.a **Ramification:** The declarations in package System and its children can be implicit. For example, since Address is not limited, the predefined "=" and "/=" operations are probably sufficient. However, the implementation is not *required* to use the predefined "=".

Implementation Advice

37 Address should be a private type.

37.a **Reason:** This promotes uniformity by avoiding having implementation-defined predefined operations for the type. We don't require it, because implementations may want to stick with what they have.

37.a.1/2 **Implementation Advice:** Type System.Address should be a private type.

37.b **Implementation Note:** It is not necessary for Address to be able to point at individual bits within a storage element. Nor is it necessary for it to be able to point at machine registers. It is intended as a memory address that matches the hardware's notion of an address.

37.c The representation of the **null** value of a general access type should be the same as that of Null_Address; instantiations of Unchecked_Conversion should work accordingly. If the implementation supports interfaces to other languages, the representation of the **null** value of a general access type should be the same as in those other languages, if appropriate.

37.d Note that the children of the Interfaces package will generally provide foreign-language-specific null values where appropriate. See UI-0065 regarding Null_Address.

NOTES

38 14 There are also some language-defined child packages of System defined elsewhere.

Extensions to Ada 83

38.a.1/1 {*extensions to Ada 83*} The declarations Max_Binary_Modulus, Max_Nonbinary_Modulus, Max_Base_Digits, Null_Address, Word_Size, Bit_Order, Default_Bit_Order, Any_Priority, Interrupt_Priority, and Default_Priority are added to System in Ada 95. The presence of ordering operators for type Address is also guaranteed (the existence of these depends on the definition of Address in an Ada 83 implementation). We do not list these as incompatibilities, as the contents of System can vary between implementations anyway; thus a program that depends on the contents of System (by using `use System;` for example) is already at risk of being incompatible when moved between Ada implementations.

Wording Changes from Ada 83

38.a Much of the content of System is standardized, to provide more uniformity across implementations. Implementations can still add their own declarations to System, but are encouraged to do so via children of System.

38.b Some of the named numbers are defined more explicitly in terms of the standard numeric types.

38.c The pragmas System_Name, Storage_Unit, and Memory_Size are no longer defined by the language. However, the corresponding declarations in package System still exist. Existing implementations may continue to support the three pragmas as implementation-defined pragmas, if they so desire.

38.d Priority semantics, including subtype Priority, have been moved to the Real Time Annex.

Extensions to Ada 95

38.e/2 {*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Type Address is defined to have preelaborable initialization, so that it can be used without restriction in preelaborated units. (If Address is defined to be a private type, as suggested by the Implementation Advice, in Ada 95 it cannot be used in some contexts in a preelaborated units. This is an unnecessary portability issue.)

38.f/2 {*AI95-00221-01*} **Amendment Correction:** Default_Bit_Order is now a static constant.

38.g/2 {*AI95-00362-01*} Package System is now Pure, so it can be portably used in more places. (Ada 95 allowed it to be Pure, but did not require that.)

13.7.1 The Package System.Storage_Elements

Static Semantics

1 The following language-defined library package exists:

2/2
```
{AI95-00362-01} package System.Storage_Elements is
   pragma Pure(Storage_Elements);
```

3
```
   type Storage_Offset is range implementation-defined;
```

```
    subtype Storage_Count is Storage_Offset range 0..Storage_Offset'Last;                4

    type Storage_Element is mod implementation-defined;                                  5
    for Storage_Element'Size use Storage_Unit;
    type Storage_Array is array
      (Storage_Offset range <>) of aliased Storage_Element;
    for Storage_Array'Component_Size use Storage_Unit;

    -- {address (arithmetic)} Address Arithmetic:                                        6

    function "+"(Left : Address; Right : Storage_Offset)                                 7
      return Address;
    function "+"(Left : Storage_Offset; Right : Address)
      return Address;
    function "-"(Left : Address; Right : Storage_Offset)
      return Address;
    function "-"(Left, Right : Address)
      return Storage_Offset;

    function "mod"(Left : Address; Right : Storage_Offset)                               8
      return Storage_Offset;

    -- Conversion to/from integers:                                                      9

    type Integer_Address is implementation-defined;                                      10
    function To_Address(Value : Integer_Address) return Address;
    function To_Integer(Value : Address) return Integer_Address;

    pragma Convention(Intrinsic, "+");                                                   11
       -- ...and so on for all language-defined subprograms declared in this package.
end System.Storage_Elements;
```

Reason: The Convention pragmas imply that the attribute Access is not allowed for those operations. 11.a

The **mod** function is needed so that the definition of Alignment makes sense. 11.b

Implementation defined: The range of Storage_Elements.Storage_Offset, the modulus of Storage_Elements.Storage_Element, and the declaration of Storage_Elements.Integer_Address.. 11.c/2

Storage_Element represents a storage element. Storage_Offset represents an offset in storage elements. Storage_Count represents a number of storage elements. {*contiguous representation* [partial]} {*discontiguous representation* [partial]} Storage_Array represents a contiguous sequence of storage elements. 12

Reason: The index subtype of Storage_Array is Storage_Offset because we wish to allow maximum flexibility. Most Storage_Arrays will probably have a lower bound of 0 or 1, but other lower bounds, including negative ones, make sense in some situations. 12.a

This paragraph was deleted.{*AI95-00114-01*} 12.b/2

Integer_Address is a [(signed or modular)] integer subtype. To_Address and To_Integer convert back and forth between this type and Address. 13

Implementation Requirements

Storage_Offset'Last shall be greater than or equal to Integer'Last or the largest possible storage offset, whichever is smaller. Storage_Offset'First shall be <= (–Storage_Offset'Last). 14

Implementation Permissions

This paragraph was deleted.{*AI95-00362-01*} 15/2

Implementation Advice

Operations in System and its children should reflect the target environment semantics as closely as is reasonable. For example, on most machines, it makes sense for address arithmetic to “wrap around.” {*Program_Error (raised by failure of run-time check)*} Operations that do not make sense should raise Program_Error. 16

Implementation Advice: Operations in System and its children should reflect the target environment; operations that do not make sense should raise Program_Error. 16.a.1/2

16.a **Discussion:** For example, on a segmented architecture, X < Y might raise Program_Error if X and Y do not point at the same segment (assuming segments are unordered). Similarly, on a segmented architecture, the conversions between Integer_Address and Address might not make sense for some values, and so might raise Program_Error.

16.b **Reason:** We considered making Storage_Element a private type. However, it is better to declare it as a modular type in the visible part, since code that uses it is already low level, and might as well have access to the underlying representation. We also considered allowing Storage_Element to be any integer type, signed integer or modular, but it is better to have uniformity across implementations in this regard, and viewing storage elements as unsigned seemed to make the most sense.

16.c **Implementation Note:** To_Address is intended for use in Address clauses. Implementations should overload To_Address if appropriate. For example, on a segmented architecture, it might make sense to have a record type representing a segment/offset pair, and have a To_Address conversion that converts from that record type to type Address.

Extensions to Ada 95

16.d/2 {*AI95-00362-01*} {*extensions to Ada 95*} Package System.Storage_Elements is now Pure, so it can be portably used in more places. (Ada 95 allowed it to be Pure, but did not require that.)

13.7.2 The Package System.Address_To_Access_Conversions

Static Semantics

1 The following language-defined generic library package exists:

2
```
generic
    type Object(<>) is limited private;
package System.Address_To_Access_Conversions is
   pragma Preelaborate(Address_To_Access_Conversions);
```

3
```
   type Object_Pointer is access all Object;
   function To_Pointer(Value : Address) return Object_Pointer;
   function To_Address(Value : Object_Pointer) return Address;
```

4
```
   pragma Convention(Intrinsic, To_Pointer);
   pragma Convention(Intrinsic, To_Address);
end System.Address_To_Access_Conversions;
```

5/2 {*AI95-00230-01*} The To_Pointer and To_Address subprograms convert back and forth between values of types Object_Pointer and Address. To_Pointer(X'Address) is equal to X'Unchecked_Access for any X that allows Unchecked_Access. To_Pointer(Null_Address) returns **null**. {*unspecified* [partial]} For other addresses, the behavior is unspecified. To_Address(**null**) returns Null_Address. To_Address(Y), where Y /= **null**, returns Y.**all**'Address.

5.a/2 **Discussion:** {*AI95-00114-01*} The programmer should ensure that the address passed to To_Pointer is either Null_Address, or the address of an object of type Object. (If Object is not a not by-reference type, the object ought to be aliased; recall that the Address attribute is not required to provide a useful result other objects.) Otherwise, the behavior of the program is unspecified; it might raise an exception or crash, for example.

5.b **Reason:** Unspecified is almost the same thing as erroneous; they both allow arbitrarily bad behavior. We don't say erroneous here, because the implementation might allow the address passed to To_Pointer to point at some memory that just happens to “look like” an object of type Object. That's not necessarily an error; it's just not portable. However, if the actual type passed to Object is (for example) an array type, the programmer would need to be aware of any dope that the implementation expects to exist, when passing an address that did not come from the Address attribute of an object of type Object.

5.c One might wonder why To_Pointer and To_Address are any better than unchecked conversions. The answer is that Address does not necessarily have the same representation as an access type. For example, an access value might point at the bounds of an array when an address would point at the first element. Or an access value might be an offset in words from someplace, whereas an address might be an offset in bytes from the beginning of memory.

Implementation Permissions

6 An implementation may place restrictions on instantiations of Address_To_Access_Conversions.

6.a **Ramification:** For example, if the hardware requires aligned loads and stores, then dereferencing an access value that is not properly aligned might raise an exception.

For another example, if the implementation has chosen to use negative component offsets (from an access value), it might not be possible to preserve the semantics, since negative offsets from the Address are not allowed. (The Address attribute always points at “the first of the storage elements....”) Note that while the implementation knows how to convert an access value into an address, it might not be able to do the reverse. To avoid generic contract model violations, the restriction might have to be detected at run time in some cases. 6.b

13.8 Machine Code Insertions

[{*machine code insertion*} A machine code insertion can be achieved by a call to a subprogram whose sequence_of_statements contains code_statements.] 1

Syntax

code_statement ::= qualified_expression; 2

A code_statement is only allowed in the handled_sequence_of_statements of a subprogram_body. If a subprogram_body contains any code_statements, then within this subprogram_body the only allowed form of statement is a code_statement (labeled or not), the only allowed declarative_items are use_clauses, and no exception_handler is allowed (comments and pragmas are allowed as usual). 3

Name Resolution Rules

{*expected type (code_statement)* [partial]} The qualified_expression is expected to be of any type. 4

Legality Rules

The qualified_expression shall be of a type declared in package System.Machine_Code. 5

Ramification: This includes types declared in children of System.Machine_Code. 5.a

A code_statement shall appear only within the scope of a with_clause that mentions package System.Machine_Code. 6

Ramification: Note that this is not a note; without this rule, it would be possible to write machine code in compilation units which depend on System.Machine_Code only indirectly. 6.a

Static Semantics

{*System.Machine_Code*} The contents of the library package System.Machine_Code (if provided) are implementation defined. The meaning of code_statements is implementation defined. [Typically, each qualified_expression represents a machine instruction or assembly directive.] 7

Discussion: For example, an instruction might be a record with an Op_Code component and other components for the operands. 7.a

Implementation defined: The contents of the visible part of package System.Machine_Code, and the meaning of code_statements. 7.b

Implementation Permissions

An implementation may place restrictions on code_statements. An implementation is not required to provide package System.Machine_Code. 8

NOTES
15 An implementation may provide implementation-defined pragmas specifying register conventions and calling conventions. 9

16 {*AI95-00318-02*} Machine code functions are exempt from the rule that a return statement is required. In fact, return statements are forbidden, since only code_statements are allowed. 10/2

Discussion: The idea is that the author of a machine code subprogram knows the calling conventions, and refers to parameters and results accordingly. The implementation should document where to put the result of a machine code function, for example, “Scalar results are returned in register 0.” 10.a

11 17 Intrinsic subprograms (see 6.3.1, “Conformance Rules”) can also be used to achieve machine code insertions. Interface to assembly language can be achieved using the features in Annex B, “Interface to Other Languages”.

Examples

12 *Example of a code statement:*

13
```
M : Mask;
procedure Set_Mask; pragma Inline(Set_Mask);
```

14
```
procedure Set_Mask is
  use System.Machine_Code; -- assume “with System.Machine_Code;” appears somewhere above
begin
  SI_Format'(Code => SSM, B => M'Base_Reg, D => M'Disp);
  -- Base_Reg and Disp are implementation-defined attributes
end Set_Mask;
```

Extensions to Ada 83

14.a {*extensions to Ada 83*} Machine code functions are allowed in Ada 95; in Ada 83, only procedures were allowed.

Wording Changes from Ada 83

14.b The syntax for code_statement is changed to say “qualified_expression” instead of “subtype_mark'record_aggregate”. Requiring the type of each instruction to be a record type is overspecification.

13.9 Unchecked Type Conversions

1 [{*unchecked type conversion*} {*type conversion (unchecked)*} {*conversion (unchecked)*} {*type_conversion: See also unchecked type conversion*} {*cast: See unchecked type conversion*} An unchecked type conversion can be achieved by a call to an instance of the generic function Unchecked_Conversion.]

Static Semantics

2 The following language-defined generic library function exists:

3
```
generic
   type Source(<>) is limited private;
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target;
pragma Convention(Intrinsic, Ada.Unchecked_Conversion);
pragma Pure(Ada.Unchecked_Conversion);
```

3.a **Reason:** The pragma Convention implies that the attribute Access is not allowed for instances of Unchecked_Conversion.

Dynamic Semantics

4 The size of the formal parameter S in an instance of Unchecked_Conversion is that of its subtype. [This is the actual subtype passed to Source, except when the actual is an unconstrained composite subtype, in which case the subtype is constrained by the bounds or discriminants of the value of the actual expression passed to S.]

5 If all of the following are true, the effect of an unchecked conversion is to return the value of an object of the target subtype whose representation is the same as that of the source object S:

6 • S'Size = Target'Size.

6.a **Ramification:** Note that there is no requirement that the Sizes be known at compile time.

7 • S'Alignment = Target'Alignment.

8 • The target subtype is not an unconstrained composite subtype.

9 • {*contiguous representation* [partial]} {*discontiguous representation* [partial]} S and the target subtype both have a contiguous representation.

- The representation of S is a representation of an object of the target subtype. 10

{*AI95-00426-01*} Otherwise, if the result type is scalar, the result of the function is implementation defined, and can have an invalid representation (see 13.9.1). If the result type is nonscalar, the effect is implementation defined; in particular, the result can be abnormal (see 13.9.1). 11/2

> **Implementation defined:** The result of unchecked conversion for instances with scalar result types whose result is not defined by the language. 11.a.1/2

> **Implementation defined:** The effect of unchecked conversion for instances with nonscalar result types whose effect is not defined by the language. 11.a/2

> **Reason:** {*AI95-00426-01*} Note the difference between these sentences; the first only says that the bits returned are implementation defined, while the latter allows any effect. The difference is because scalar objects should never be abnormal unless their assignment was disrupted or if they are a subcomponent of an abnormal composite object. Neither exception applies to instances of Unchecked_Conversion. 11.a.1/2

> **Ramification:** {*AI95-00426-01*} Whenever unchecked conversions are used, it is the programmer's responsibility to ensure that these conversions maintain the properties that are guaranteed by the language for objects of the target type. For nonscalar types, this requires the user to understand the underlying run-time model of the implementation. The execution of a program that violates these properties by means of unchecked conversions returning a nonscalar type is erroneous. Properties of scalar types can be checked by using the Valid attribute (see 13.9.2); programs can avoid violating properties of the type (and erroneous execution) by careful use of this attribute. 11.a.2/2

> An instance of Unchecked_Conversion can be applied to an object of a private type, assuming the implementation allows it. 11.b

Implementation Permissions

An implementation may return the result of an unchecked conversion by reference, if the Source type is not a by-copy type. [In this case, the result of the unchecked conversion represents simply a different (read-only) view of the operand of the conversion.] 12

> **Ramification:** In other words, the result object of a call on an instance of Unchecked_Conversion can occupy the same storage as the formal parameter S. 12.a

An implementation may place restrictions on Unchecked_Conversion. 13

> **Ramification:** For example, an instantiation of Unchecked_Conversion for types for which unchecked conversion doesn't make sense may be disallowed. 13.a

Implementation Advice

{*AI95-00051-02*} Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data. 14/2

> **Implementation Advice:** Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data in an instance of Unchecked_Conversion. 14.a.1/2

> **Ramification:** On the other hand, we have no advice to offer about discriminants and tag fields. 14.a

The implementation should not generate unnecessary run-time checks to ensure that the representation of S is a representation of the target type. It should take advantage of the permission to return by reference when possible. Restrictions on unchecked conversions should be avoided unless required by the target environment. 15

> **Implementation Advice:** There should not be unnecessary run-time checks on the result of an Unchecked_Conversion; the result should be returned by reference when possible. Restrictions on Unchecked_Conversions should be avoided. 15.a.1/2

> **Implementation Note:** As an example of an unnecessary run-time check, consider a record type with gaps between components. The compiler might assume that such gaps are always zero bits. If a value is produced that does not obey that assumption, then the program might misbehave. The implementation should not generate extra code to check for zero bits (except, perhaps, in a special error-checking mode). 15.a

{*recommended level of support (unchecked conversion)* [partial]} The recommended level of support for unchecked conversions is: 16

17 • Unchecked conversions should be supported and should be reversible in the cases where this clause defines the result. {*contiguous representation* [partial]} {*discontiguous representation* [partial]} To enable meaningful use of unchecked conversion, a contiguous representation should be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in this paragraph, and for record subtypes without discriminants whose component subtypes are described in this paragraph.

17.a/2 **Implementation Advice:** The recommended level of support for Unchecked_Conversion should be followed.

Wording Changes from Ada 95

17.b/2 {*AI95-00051-02*} The implementation advice about the size of array objects was moved to 13.3 so that all of the advice about Size is in one place.

17.c/2 {*AI95-00426-01*} Clarified that the result of Unchecked_Conversion for scalar types can be invalid, but not abnormal.

13.9.1 Data Validity

1 Certain actions that can potentially lead to erroneous execution are not directly erroneous, but instead can cause objects to become *abnormal*. Subsequent uses of abnormal objects can be erroneous.

2 A scalar object can have an *invalid representation*, which means that the object's representation does not represent any value of the object's subtype. {*uninitialized variables* [distributed]} The primary cause of invalid representations is uninitialized variables.

3 Abnormal objects and invalid representations are explained in this subclause.

Dynamic Semantics

4 {*normal state of an object* [distributed]} {*abnormal state of an object* [distributed]} When an object is first created, and any explicit or default initializations have been performed, the object and all of its parts are in the *normal* state. Subsequent operations generally leave them normal. However, an object or part of an object can become *abnormal* in the following ways:

5 • {*disruption of an assignment*} An assignment to the object is disrupted due to an abort (see 9.8) or due to the failure of a language-defined check (see 11.6).

6/2 • {*AI95-00426-01*} The object is not scalar, and is passed to an **in out** or **out** parameter of an imported procedure, the Read procedure of an instance of Sequential_IO, Direct_IO, or Storage_IO, or the stream attribute T'Read, if after return from the procedure the representation of the parameter does not represent a value of the parameter's subtype.

6.1/2 • {*AI95-00426-01*} The object is the return object of a function call of a nonscalar type, and the function is an imported function, an instance of Unchecked_Conversion, or the stream attribute T'Input, if after return from the function the representation of the return object does not represent a value of the function's subtype.

6.a/2 **Discussion:** We explicitly list the routines involved in order to avoid future arguments. All possibilities are listed.

6.b/2 We did not include Stream_IO.Read in the list above. A Stream_Element should include all possible bit patterns, and thus it cannot be invalid. Therefore, the parameter will always represent a value of its subtype. By omitting this routine, we make it possible to write arbitrary I/O operations without any possibility of abnormal objects.

6.2/2 {*AI95-00426-01*} [For an imported object, it is the programmer's responsibility to ensure that the object remains in a normal state.]

6.c/2 **Proof:** This follows (and echos) the standard rule of interfacing; the programmer must ensure that Ada semantics are followed (see B.1).

7 {*unspecified* [partial]} Whether or not an object actually becomes abnormal in these cases is not specified. An abnormal object becomes normal again upon successful completion of an assignment to the object as a whole.

Erroneous Execution

8 {*erroneous execution (cause)* [partial]} It is erroneous to evaluate a primary that is a name denoting an abnormal object, or to evaluate a prefix that denotes an abnormal object.

8.a/2 *This paragraph was deleted.*{*AI95-00114-01*}

8.b **Ramification:** The **in out** or **out** parameter case does not apply to scalars; bad scalars are merely invalid representations, rather than abnormal, in this case.

8.c/2 **Reason:** {*AI95-00114-01*} The reason we allow access objects, and objects containing subcomponents of an access type, to become abnormal is because the correctness of an access value cannot necessarily be determined merely by looking at the bits of the object. The reason we allow scalar objects to become abnormal is that we wish to allow the compiler to optimize assuming that the value of a scalar object belongs to the object's subtype, if the compiler can prove that the object is initialized with a value that belongs to the subtype. The reason we allow composite objects to become abnormal is that such object might be represented with implicit levels of indirection; if those are corrupted, then even assigning into a component of the object, or simply asking for its Address, might have an unpredictable effect. The same is true if the discriminants have been destroyed.

Bounded (Run-Time) Errors

9 {*invalid representation*} {*bounded error (cause)* [partial]} If the representation of a scalar object does not represent a value of the object's subtype (perhaps because the object was not initialized), the object is said to have an *invalid representation*. It is a bounded error to evaluate the value of such an object. {*Program_Error (raised by failure of run-time check)*} {*Constraint_Error (raised by failure of run-time check)*} If the error is detected, either Constraint_Error or Program_Error is raised. Otherwise, execution continues using the invalid representation. The rules of the language outside this subclause assume that all objects have valid representations. The semantics of operations on invalid representations are as follows:

9.a **Discussion:** The AARM is more explicit about what happens when the value of the case expression is an invalid representation.

9.b/2 **Ramification:** {*AI95-00426-01*} This includes the result object of functions, including the result of Unchecked_Conversion, T'Input, and imported functions.

10 • If the representation of the object represents a value of the object's type, the value of the type is used.

11 • If the representation of the object does not represent a value of the object's type, the semantics of operations on such representations is implementation-defined, but does not by itself lead to erroneous or unpredictable execution, or to other objects becoming abnormal.

11.a/2 **Implementation Note:** {*AI95-00426-01*} This means that the implementation must take care not to use an invalid representation in a way that might cause erroneous execution. For instance, the exception mandated for case_statements must be raised. Array indexing must not cause memory outside of the array to be written (and usually, not read either). These cases and similar cases may require explicit checks by the implementation.

Erroneous Execution

12/2 {*AI95-00167-01*} {*erroneous execution (cause)* [partial]} A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, the result object has an invalid representation, and the result is used other than as the expression of an assignment_statement or an object_declaration, or as the prefix of a Valid attribute. If such a result object is used as the source of an assignment, and the assigned value is an invalid representation for the target of the assignment, then any use of the target object prior to a further assignment to the target object, other than as the prefix of a Valid attribute reference, is erroneous.

12.a/2 **Ramification:** {*AI95-00167-01*} In a typical implementation, every bit pattern that fits in an object of a signed integer subtype will represent a value of the type, if not of the subtype. However, for an enumeration or floating point type, as well as some modular types, there are typically bit patterns that do not represent any value of the type. In such cases, the implementation ought to define the semantics of operations on the invalid representations in the obvious manner (assuming the bounded error is not detected): a given representation should be equal to itself, a representation that is in between the internal codes of two enumeration literals should behave accordingly when passed to comparison operators and membership tests, etc. We considered *requiring* such sensible behavior, but it resulted in too much arcane verbiage, and since implementations have little incentive to behave irrationally, such verbiage is not important to have.

12.b/2 {*AI95-00167-01*} If a stand-alone scalar object is initialized to a an in-range value, then the implementation can take advantage of the fact that the use of any out-of-range value has to be erroneous. Such an out-of-range value can be produced only by things like unchecked conversion, imported functions, and abnormal values caused by disruption of an assignment due to abort or to failure of a language-defined check. This depends on out-of-range values being checked before assignment (that is, checks are not optimized away unless they are proven redundant).

12.c Consider the following example:

12.d/2
```
{AI95-00167-01} type My_Int is range 0..99;
function Safe_Convert is new Unchecked_Conversion(My_Int, Integer);
function Unsafe_Convert is new Unchecked_Conversion(My_Int, Positive);
X : Positive := Safe_Convert(0); -- Raises Constraint_Error.
Y : Positive := Unsafe_Convert(0); -- Bounded Error, may be invalid.
B : Boolean := Y'Valid; -- OK, B = False.
Z : Positive := Y+1; -- Erroneous to use Y.
```

12.e/2 {*AI95-00167-01*} {*AI95-00426-01*} The call to Unsafe_Convert is a bounded error, which might raise Constraint_Error, Program_Error, or return an invalid value. Moreover, if an exception is not raised, most uses of that invalid value (including the use of Y) cause erroneous execution. The call to Safe_Convert is not erroneous. The result object is an object of subtype Integer containing the value 0. The assignment to X is required to do a constraint check; the fact that the conversion is unchecked does not obviate the need for subsequent checks required by the language rules.

12.e.1/2 {*AI95-00167-01*} {*AI95-00426-01*} The reason for delaying erroneous execution until the object is used is so that the invalid representation can be tested for validity using the Valid attribute (see 13.9.2) without causing execution to become erroneous. Note that this delay does not imply an exception will not be raised; an implementation could treat both conversions in the example in the same way and raise Constraint_Error.

12.f **Implementation Note:** If an implementation wants to have a “friendly” mode, it might always assign an uninitialized scalar a default initial value that is outside the object's subtype (if there is one), and check for this value on some or all reads of the object, so as to help detect references to uninitialized scalars. Alternatively, an implementation might want to provide an “unsafe” mode where it presumed even uninitialized scalars were always within their subtype.

12.g **Ramification:** The above rules imply that it is a bounded error to apply a predefined operator to an object with a scalar subcomponent having an invalid representation, since this implies reading the value of each subcomponent. Either Program_Error or Constraint_Error is raised, or some result is produced, which if composite, might have a corresponding scalar subcomponent still with an invalid representation.

12.h Note that it is not an error to assign, convert, or pass as a parameter a composite object with an uninitialized scalar subcomponent. In the other hand, it is a (bounded) error to apply a predefined operator such as =, <, and **xor** to a composite operand with an invalid scalar subcomponent.

13 {*erroneous execution (cause)* [partial]} The dereference of an access value is erroneous if it does not designate an object of an appropriate type or a subprogram with an appropriate profile, if it designates a nonexistent object, or if it is an access-to-variable value that designates a constant object. [Such an access value can exist, for example, because of Unchecked_Deallocation, Unchecked_Access, or Unchecked_Conversion.]

13.a **Ramification:** The above mentioned Unchecked_... features are not the only causes of such access values. For example, interfacing to other languages can also cause the problem.

13.b One obscure example is if the Adjust subprogram of a controlled type uses Unchecked_Access to create an access-to-variable value designating a subcomponent of its controlled parameter, and saves this access value in a global object. When Adjust is called during the initialization of a constant object of the type, the end result will be an access-to-variable value that designates a constant object.

NOTES

14 18 Objects can become abnormal due to other kinds of actions that directly update the object's representation; such actions are generally considered directly erroneous, however.

Wording Changes from Ada 83

14.a In order to reduce the amount of erroneousness, we separate the concept of an undefined value into objects with invalid representation (scalars only) and abnormal objects.

14.b Reading an object with an invalid representation is a bounded error rather than erroneous; reading an abnormal object is still erroneous. In fact, the only safe thing to do to an abnormal object is to assign to the object as a whole.

Wording Changes from Ada 95

{*AI95-00167-01*} The description of erroneous execution for Unchecked_Conversion and imported objects was tightened up so that using the Valid attribute to test such a value is not erroneous. 14.c/2

{*AI95-00426-01*} Clarified the definition of objects that can become abnormal; made sure that all of the possibilities are included. 14.d/2

13.9.2 The Valid Attribute

The Valid attribute can be used to check the validity of data produced by unchecked conversion, input, interface to foreign languages, and the like. 1

Static Semantics

For a prefix X that denotes a scalar object [(after any implicit dereference)], the following attribute is defined: 2

X'Valid Yields True if and only if the object denoted by X is normal and has a valid representation. The value of this attribute is of the predefined type Boolean. 3

Ramification: Having checked that X'Valid is True, it is safe to read the value of X without fear of erroneous execution caused by abnormality, or a bounded error caused by an invalid representation. Such a read will produce a value in the subtype of X. 3.a

NOTES

19 Invalid data can be created in the following cases (not counting erroneous or unpredictable execution): 4

- an uninitialized scalar object, 5
- the result of an unchecked conversion, 6
- input, 7
- interface to another language (including machine code), 8
- aborting an assignment, 9
- disrupting an assignment due to the failure of a language-defined check (see 11.6), and 10
- use of an object whose Address has been specified. 11

20 X'Valid is not considered to be a read of X; hence, it is not an error to check the validity of invalid data. 12

21 {*AI95-00426-01*} The Valid attribute may be used to check the result of calling an instance of Unchecked_Conversion (or any other operation that can return invalid values). However, an exception handler should also be provided because implementations are permitted to raise Constraint_Error or Program_Error if they detect the use of an invalid representation (see 13.9.1). 13/2

Ramification: If X is of an enumeration type with a representation clause, then X'Valid checks that the value of X when viewed as an integer is one of the specified internal codes. 13.a

Reason: Valid is defined only for scalar objects because the implementation and description burden would be too high for other types. For example, given a typical run-time model, it is impossible to check the validity of an access value. The same applies to composite types implemented with internal pointers. One can check the validity of a composite object by checking the validity of each of its scalar subcomponents. The user should ensure that any composite types that need to be checked for validity are represented in a way that does not involve implementation-defined components, or gaps between components. Furthermore, such types should not contain access subcomponents. 13.b

This paragraph was deleted.{*AI95-00114-01*} 13.c/2

Extensions to Ada 83

{*extensions to Ada 83*} X'Valid is new in Ada 95. 13.d

Wording Changes from Ada 95

{*AI95-00426-01*} Added a note explaining that handlers for Constraint_Error and Program_Error are needed in the general case of testing for validity. (An implementation could document cases where these are not necessary, but there is no language requirement. 13.e/2

13.10 Unchecked Access Value Creation

1 [The attribute Unchecked_Access is used to create access values in an unsafe manner — the programmer is responsible for preventing “dangling references.”]

Static Semantics

2 The following attribute is defined for a prefix X that denotes an aliased view of an object:

3 X'Unchecked_Access

All rules and semantics that apply to X'Access (see 3.10.2) apply also to X'Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package. {*Access attribute: See also Unchecked_Access attribute*}

NOTES

4 22 This attribute is provided to support the situation where a local object is to be inserted into a global linked data structure, when the programmer knows that it will always be removed from the data structure prior to exiting the object's scope. The Access attribute would be illegal in this case (see 3.10.2, “Operations of Access Types”).

4.a **Ramification:** {*expected type (Unchecked_Access attribute)* [partial]} The expected type for X'Unchecked_Access is as for X'Access.

4.b If an attribute_reference with Unchecked_Access is used as the actual parameter for an access parameter, an Accessibility_Check can never fail on that access parameter.

5 23 There is no Unchecked_Access attribute for subprograms.

5.a/2 **Reason:** {*AI95-00254-01*} Such an attribute would allow unsafe “downward closures”, where an access value designating a more nested subprogram is passed to a less nested subprogram. (Anonymous access-to-subprogram parameters provide safe “downward closures”.) This requires some means of reconstructing the global environment for the more nested subprogram, so that it can do up-level references to objects. The two methods of implementing up-level references are displays and static links. If unsafe downward closures were supported, each access-to-subprogram value would have to carry the static link or display with it. We don't want to require the space and time overhead of requiring the extra information for all access-to-subprogram types, especially as including it would make interfacing to other languages (like C) harder.

5.b If desired, an instance of Unchecked_Conversion can be used to create an access value of a global access-to-subprogram type that designates a local subprogram. The semantics of using such a value are not specified by the language. In particular, it is not specified what happens if such subprograms make up-level references; even if the frame being referenced still exists, the up-level reference might go awry if the representation of a value of a global access-to-subprogram type doesn't include a static link.

13.11 Storage Management

1 [{*user-defined storage management*} {*storage management (user-defined)*} {*user-defined heap management*} {*heap management (user-defined)*} Each access-to-object type has an associated storage pool. The storage allocated by an allocator comes from the pool; instances of Unchecked_Deallocation return storage to the pool. Several access types can share the same pool.]

2/2 {*AI95-00435-01*} [A storage pool is a variable of a type in the class rooted at Root_Storage_Pool, which is an abstract limited controlled type. By default, the implementation chooses a *standard storage pool* for each access-to-object type. The user may define new pool types, and may override the choice of pool for an access-to-object type by specifying Storage_Pool for the type.]

2.a **Ramification:** By default, the implementation might choose to have a single global storage pool, which is used (by default) by all access types, which might mean that storage is reclaimed automatically only upon partition completion. Alternatively, it might choose to create a new pool at each accessibility level, which might mean that storage is reclaimed for an access type when leaving the appropriate scope. Other schemes are possible.

Legality Rules

3 If Storage_Pool is specified for a given access type, Storage_Size shall not be specified for it.

Reason: The Storage_Pool determines the Storage_Size; hence it would not make sense to specify both. Note that this rule is simplified by the fact that the aspects in question cannot be specified for derived types, nor for non-first subtypes, so we don't have to worry about whether, say, Storage_Pool on a derived type overrides Storage_Size on the parent type. For the same reason, “specified” means the same thing as “directly specified” here. 3.a

Static Semantics

The following language-defined library package exists: 4

```
with Ada.Finalization;                                                5
with System.Storage_Elements;
package System.Storage_Pools is
    pragma Preelaborate(System.Storage_Pools);

{AI95-00161-01}     type Root_Storage_Pool is                         6/2
        abstract new Ada.Finalization.Limited_Controlled with private;
    pragma Preelaborable_Initialization(Root_Storage_Pool);

    procedure Allocate(                                               7
      Pool : in out Root_Storage_Pool;
      Storage_Address : out Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count) is abstract;

    procedure Deallocate(                                             8
      Pool : in out Root_Storage_Pool;
      Storage_Address : in Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count) is abstract;

    function Storage_Size(Pool : Root_Storage_Pool)                   9
        return Storage_Elements.Storage_Count is abstract;

private                                                               10
   ... -- not specified by the language
end System.Storage_Pools;
```

Reason: The Alignment parameter is provided to Deallocate because some allocation strategies require it. If it is not needed, it can be ignored. 10.a

{*storage pool type*} {*pool type*} A *storage pool type* (or *pool type*) is a descendant of Root_Storage_Pool. {*storage pool element*} {*pool element*} {*element (of a storage pool)*} The *elements* of a storage pool are the objects allocated in the pool by allocators. 11

Discussion: In most cases, an element corresponds to a single memory block allocated by Allocate. However, in some cases the implementation may choose to associate more than one memory block with a given pool element. 11.a

{*8652/0009*} {*AI95-00137-01*} {*AI95-00435-01*} For every access-to-object subtype S, the following representation attributes are defined: 12/2

S'Storage_Pool 13

Denotes the storage pool of the type of S. The type of this attribute is Root_Storage_-Pool'Class.

S'Storage_Size 14

Yields the result of calling Storage_Size(S'Storage_Pool)[, which is intended to be a measure of the number of storage elements reserved for the pool.] The type of this attribute is *universal_integer*.

Ramification: Storage_Size is also defined for task subtypes and objects — see 13.3. 14.a

Storage_Size is not a measure of how much un-allocated space is left in the pool. That is, it includes both allocated and unallocated space. Implementations and users may provide a Storage_Available function for their pools, if so desired. 14.b

{*specifiable (of Storage_Size for a non-derived access-to-object type)* [partial]} {*specifiable (of Storage_Pool for a non-derived access-to-object type)* [partial]} {*Storage_Pool clause*} {*Storage_Size clause*} Storage_Size or Storage_Pool may be specified for a non-derived access-to-object type via an attribute_definition_-clause; the name in a Storage_Pool clause shall denote a variable. 15

16 An allocator of type T allocates storage from T's storage pool. If the storage pool is a user-defined object, then the storage is allocated by calling Allocate, passing T'Storage_Pool as the Pool parameter. The Size_In_Storage_Elements parameter indicates the number of storage elements to be allocated, and is no more than D'Max_Size_In_Storage_Elements, where D is the designated subtype. The Alignment parameter is D'Alignment. {*contiguous representation* [partial]} {*discontiguous representation* [partial]} The result returned in the Storage_Address parameter is used by the allocator as the address of the allocated storage, which is a contiguous block of memory of Size_In_Storage_Elements storage elements. [Any exception propagated by Allocate is propagated by the allocator.]

16.a **Ramification:** If the implementation chooses to represent the designated subtype in multiple pieces, one allocator evaluation might result in more than one call upon Allocate. In any case, allocators for the access type obtain all the required storage for an object of the designated type by calling the specified Allocate procedure.

16.b Note that the implementation does not turn other exceptions into Storage_Error.

16.b.1/1 {*8652/0111*} {*AI95-00103-01*} If D (the designated type of T) includes subcomponents of other access types, they will be allocated from the storage pools for those types, even if those allocators are executed as part of the allocator of T (as part of the initialization of the object). For instance, an access-to-task type TT may allocate the data structures used to implement the task value from other storage pools. (In particular, the task stack does not necessarily need to be allocated from the storage pool for TT.)

17 {*standard storage pool*} If Storage_Pool is not specified for a type defined by an access_to_object_definition, then the implementation chooses a standard storage pool for it in an implementation-defined manner. {*Storage_Check* [partial]} {*check, language-defined (Storage_Check)*} {*Storage_Error (raised by failure of run-time check)*} In this case, the exception Storage_Error is raised by an allocator if there is not enough storage. It is implementation defined whether or not the implementation provides user-accessible names for the standard pool type(s).

17.a/2 *This paragraph was deleted.*

17.a.1/2 **Discussion:** The manner of choosing a storage pool is covered by a Documentation Requirement below, so it is not summarized here.

17.b **Implementation defined:** Whether or not the implementation provides user-accessible names for the standard pool type(s).

17.c/2 **Ramification:** {*AI95-00230-01*} An access-to-object type defined by a derived_type_definition inherits its pool from its parent type, so all access-to-object types in the same derivation class share the same pool. Hence the “defined by an access_to_object_definition” wording above.

17.d {*contiguous representation* [partial]} {*discontiguous representation* [partial]} There is no requirement that all storage pools be implemented using a contiguous block of memory (although each allocation returns a pointer to a contiguous block of memory).

18 If Storage_Size is specified for an access type, then the Storage_Size of this pool is at least that requested, and the storage for the pool is reclaimed when the master containing the declaration of the access type is left. {*Storage_Error (raised by failure of run-time check)*} If the implementation cannot satisfy the request, Storage_Error is raised at the point of the attribute_definition_clause. If neither Storage_Pool nor Storage_Size are specified, then the meaning of Storage_Size is implementation defined.

18.a/2 **Implementation defined:** The meaning of Storage_Size when neither the Storage_Size nor the Storage_Pool is specified for an access type.

18.b **Ramification:** The Storage_Size function and attribute will return the actual size, rather than the requested size. Comments about rounding up, zero, and negative on task Storage_Size apply here, as well. See also AI83-00557, AI83-00558, and AI83-00608.

18.c The expression in a Storage_Size clause need not be static.

18.d The reclamation happens after the master is finalized.

18.e **Implementation Note:** For a pool allocated on the stack, normal stack cut-back can accomplish the reclamation. For a library-level pool, normal partition termination actions can accomplish the reclamation.

19 If Storage_Pool is specified for an access type, then the specified pool is used.

20 {*unspecified* [partial]} The effect of calling Allocate and Deallocate for a standard storage pool directly (rather than implicitly via an allocator or an instance of Unchecked_Deallocation) is unspecified.

20.a **Ramification:** For example, an allocator might put the pool element on a finalization list. If the user directly Deallocates it, instead of calling an instance of Unchecked_Deallocation, then the implementation would probably try to finalize the object upon master completion, which would be bad news. Therefore, the implementation should define such situations as erroneous.

Erroneous Execution

21 {*erroneous execution (cause)* [partial]} If Storage_Pool is specified for an access type, then if Allocate can satisfy the request, it should allocate a contiguous block of memory, and return the address of the first storage element in Storage_Address. The block should contain Size_In_Storage_Elements storage elements, and should be aligned according to Alignment. The allocated storage should not be used for any other purpose while the pool element remains in existence. If the request cannot be satisfied, then Allocate should propagate an exception [(such as Storage_Error)]. If Allocate behaves in any other manner, then the program execution is erroneous.

Documentation Requirements

22 An implementation shall document the set of values that a user-defined Allocate procedure needs to accept for the Alignment parameter. An implementation shall document how the standard storage pool is chosen, and how storage is allocated by standard storage pools.

22.a/2 *This paragraph was deleted.*

22.b/2 **Documentation Requirement:** The set of values that a user-defined Allocate procedure needs to accept for the Alignment parameter. How the standard storage pool is chosen, and how storage is allocated by standard storage pools.

Implementation Advice

23 An implementation should document any cases in which it dynamically allocates heap storage for a purpose other than the evaluation of an allocator.

23.a.1/2 **Implementation Advice:** Any cases in which heap storage is dynamically allocated other than as part of the evaluation of an allocator should be documented.

23.a **Reason:** This is “Implementation Advice” because the term “heap storage” is not formally definable; therefore, it is not testable whether the implementation obeys this advice.

24 A default (implementation-provided) storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects.

24.a.1/2 **Implementation Advice:** A default storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects.

24.a **Ramification:** Unchecked_Deallocation is not defined for such types. If the access-to-constant type is library-level, then no deallocation (other than at partition completion) will ever be necessary, so if the size needed by an allocator of the type is known at link-time, then the allocation should be performed statically. If, in addition, the initial value of the designated object is known at compile time, the object can be allocated to read-only memory.

24.b **Implementation Note:** If the Storage_Size for an access type is specified, the storage pool should consist of a contiguous block of memory, possibly allocated on the stack. The pool should contain approximately this number of storage elements. These storage elements should be reserved at the place of the Storage_Size clause, so that allocators cannot raise Storage_Error due to running out of pool space until the appropriate number of storage elements has been used up. This approximate (possibly rounded-up) value should be used as a maximum; the implementation should not increase the size of the pool on the fly. If the Storage_Size for an access type is specified as zero, then the pool should not take up any storage space, and any allocator for the type should raise Storage_Error.

24.c **Ramification:** Note that most of this is approximate, and so cannot be (portably) tested. That's why we make it an Implementation Note. There is no particular number of allocations that is guaranteed to succeed, and there is no particular number of allocations that is guaranteed to fail.

25/2 {*AI95-00230-01*} The storage pool used for an allocator of an anonymous access type should be determined as follows:

25.1/2 • {*AI95-00230-01*} {*AI95-00416-01*} If the allocator is defining a coextension (see 3.10.2) of an object being created by an outer allocator, then the storage pool used for the outer allocator should also be used for the coextension;

25.2/2 • {*AI95-00230-01*} For other access discriminants and access parameters, the storage pool should be created at the point of the allocator, and be reclaimed when the allocated object becomes inaccessible;

25.3/2 • {*AI95-00230-01*} Otherwise, a default storage pool should be created at the point where the anonymous access type is elaborated; such a storage pool need not support deallocation of individual objects.

25.a.1/2 **Implementation Advice:** Usually, a storage pool for an access discriminant or access parameter should be created at the point of an allocator, and be reclaimed when the designated object becomes inaccessible. For other anonymous access types, the pool should be created at the point where the type is elaborated and need not support deallocation of individual objects.

25.a/2 **Implementation Note:** {*AI95-00230-01*} For access parameters and access discriminants, the "storage pool" for an anonymous access type would not normally exist as a separate entity. Instead, the designated object of the allocator would be allocated, in the case of an access parameter, as a local aliased variable at the call site, and in the case of an access discriminant, contiguous with the object containing the discriminant. This is similar to the way storage for aggregates is typically managed.

25.b/2 {*AI95-00230-01*} For other sorts of anonymous access types, this implementation is not possible in general, as the accessibility of the anonymous access type is that of its declaration, while the allocator could be more nested. In this case, a "real" storage pool is required. Note, however, that this storage pool need not support (separate) deallocation, as it is not possible to instantiate Unchecked_Deallocation with an anonymous access type. (If deallocation is needed, the object should be allocated for a named access type and converted.) Thus, deallocation only need happen when the anonymous access type itself goes out of scope; this is similar to the case of an access-to-constant type.

NOTES

26 24 A user-defined storage pool type can be obtained by extending the Root_Storage_Pool type, and overriding the primitive subprograms Allocate, Deallocate, and Storage_Size. A user-defined storage pool can then be obtained by declaring an object of the type extension. The user can override Initialize and Finalize if there is any need for non-trivial initialization and finalization for a user-defined pool type. For example, Finalize might reclaim blocks of storage that are allocated separately from the pool object itself.

27 25 The writer of the user-defined allocation and deallocation procedures, and users of allocators for the associated access type, are responsible for dealing with any interactions with tasking. In particular:

28 • If the allocators are used in different tasks, they require mutual exclusion.

29 • If they are used inside protected objects, they cannot block.

30 • If they are used by interrupt handlers (see C.3, "Interrupt Support"), the mutual exclusion mechanism has to work properly in that context.

31 26 The primitives Allocate, Deallocate, and Storage_Size are declared as abstract (see 3.9.3), and therefore they have to be overridden when a new (non-abstract) storage pool type is declared.

31.a **Ramification:** Note that the Storage_Pool attribute denotes an object, rather than a value, which is somewhat unusual for attributes.

31.b The calls to Allocate, Deallocate, and Storage_Size are dispatching calls — this follows from the fact that the actual parameter for Pool is T'Storage_Pool, which is of type Root_Storage_Pool'Class. In many cases (including all cases in which Storage_Pool is not specified), the compiler can determine the tag statically. However, it is possible to construct cases where it cannot.

31.c All access types in the same derivation class share the same pool, whether implementation defined or user defined. This is necessary because we allow type conversions among them (even if they are pool-specific), and we want pool-specific access values to always designate an element of the right pool.

31.d **Implementation Note:** If an access type has a standard storage pool, then the implementation doesn't actually have to follow the pool interface described here, since this would be semantically invisible. For example, the allocator could conceivably be implemented with inline code.

Examples

32 To associate an access type with a storage pool object, the user first declares a pool object of some type derived from Root_Storage_Pool. Then, the user defines its Storage_Pool attribute, as follows:

```
Pool_Object : Some_Storage_Pool_Type;                                   33
```

```
type T is access Designated;                                            34
for T'Storage_Pool use Pool_Object;
```

Another access type may be added to an existing storage pool, via: 35

```
for T2'Storage_Pool use T'Storage_Pool;                                 36
```

The semantics of this is implementation defined for a standard storage pool. 37

Reason: For example, the implementation is allowed to choose a storage pool for T that takes advantage of the fact that T is of a certain size. If T2 is not of that size, then the above will probably not work. 37.a

As usual, a derivative of Root_Storage_Pool may define additional operations. For example, presuming that Mark_Release_Pool_Type has two additional operations, Mark and Release, the following is a possible use: 38

```
{8652/0041} {AI95-00066-01} type Mark_Release_Pool_Type                39/1
   (Pool_Size : Storage_Elements.Storage_Count;
    Block_Size : Storage_Elements.Storage_Count)
        is new Root_Storage_Pool with private;
```

```
...                                                                     40
```

```
MR_Pool : Mark_Release_Pool_Type (Pool_Size => 2000,                    41
                                  Block_Size => 100);
```

```
type Acc is access ...;                                                 42
for Acc'Storage_Pool use MR_Pool;
...
```

```
Mark(MR_Pool);                                                          43
... -- Allocate objects using “new Designated(...)”.
Release(MR_Pool); -- Reclaim the storage.
```

Extensions to Ada 83

{*extensions to Ada 83*} User-defined storage pools are new to Ada 95. 43.a

Wording Changes from Ada 83

Ada 83 had a concept called a “collection,” which is similar to what we call a storage pool. All access types in the same derivation class shared the same collection. In Ada 95, all access types in the same derivation class share the same storage pool, but other (unrelated) access types can also share the same storage pool, either by default, or as specified by the user. A collection was an amorphous collection of objects; a storage pool is a more concrete concept — hence the different name. 43.b

RM83 states the erroneousness of reading or updating deallocated objects incorrectly by missing various cases. 43.c

Incompatibilities With Ada 95

{*AI95-00435-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** Storage pools (and Storage_Size) are not defined for access-to-subprogram types. The original Ada 95 wording defined the attributes, but said nothing about their values. If a program uses attributes Storage_Pool or Storage_Size on an access-to-subprogram type, it will need to be corrected for Ada 2005. That's a good thing, as such a use is a bug — the concepts never were defined for such types. 43.d/2

Extensions to Ada 95

{*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Added pragma Preelaborable_Initialization to type Root_Storage_Pool, so that extensions of it can be used to declare default-initialized objects in preelaborated units. 43.e/2

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added wording to specify that these are representation attributes. 43.f/2

{*AI95-00230-01*} {*AI95-00416-01*} Added wording to clarify that an allocator for a coextension nested inside an outer allocator shares the pool with the outer allocator. 43.g/2

13.11.1 The Max_Size_In_Storage_Elements Attribute

1 [The Max_Size_In_Storage_Elements attribute is useful in writing user-defined pool types.]

Static Semantics

2 For every subtype S, the following attribute is defined:

3/2 S'Max_Size_In_Storage_Elements

{*AI95-00256-01*} {*AI95-00416-01*} Denotes the maximum value for Size_In_Storage_Elements that could be requested by the implementation via Allocate for an access type whose designated subtype is S. For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then this accounts for any calls on Allocate that could be performed to provide space for such coextensions. The value of this attribute is of type *universal_integer*.

3.a **Ramification:** If S is an unconstrained array subtype, or an unconstrained subtype with discriminants, S'Max_Size_In_Storage_Elements might be very large.

Wording Changes from Ada 95

3.b/2 {*AI95-00256-01*} Corrected the wording so that a fortune-telling compiler that can see the future execution of the program is not required.

13.11.2 Unchecked Storage Deallocation

1 [{*unchecked storage deallocation*} {*storage deallocation (unchecked)*} {*deallocation of storage*} {*reclamation of storage*} {*freeing storage*} Unchecked storage deallocation of an object designated by a value of an access type is achieved by a call to an instance of the generic procedure Unchecked_Deallocation.]

Static Semantics

2 The following language-defined generic library procedure exists:

3
```
generic
   type Object(<>) is limited private;
   type Name   is access  Object;
procedure Ada.Unchecked_Deallocation(X : in out Name);
pragma Convention(Intrinsic, Ada.Unchecked_Deallocation);
pragma Preelaborate(Ada.Unchecked_Deallocation);
```

3.a **Reason:** The pragma Convention implies that the attribute Access is not allowed for instances of Unchecked_Deallocation.

Dynamic Semantics

4 Given an instance of Unchecked_Deallocation declared as follows:

5
```
procedure Free is
    new Ada.Unchecked_Deallocation(
        object_subtype_name, access_to_variable_subtype_name);
```

6 Procedure Free has the following effect:

7 1. After executing Free(X), the value of X is **null**.

8 2. Free(X), when X is already equal to **null**, has no effect.

9/2 3. {*AI95-00416-01*} Free(X), when X is not equal to **null** first performs finalization of the object designated by X (and any coextensions of the object — see 3.10.2), as described in 7.6.1. It then deallocates the storage occupied by the object designated by X (and any coextensions). If the storage pool is a user-defined object, then the storage is deallocated by calling Deallocate, passing *access_to_variable_subtype_name*'Storage_Pool as the Pool parameter. Storage_Address is the value returned in the Storage_Address parameter of the corresponding Allocate call. Size_In_-Storage_Elements and Alignment are the same values passed to the corresponding Allocate call. There is one exception: if the object being freed contains tasks, the object might not be deallocated.

9.a **Ramification:** Free calls only the specified Deallocate procedure to do deallocation. For any given object deallocation, the number of calls to Free (usually one) will be equal to the number of Allocate calls it took to allocate the object. We do not define the relative order of multiple calls used to deallocate the same object — that is, if the allocator allocated two pieces *x* and *y*, then Free might deallocate *x* and then *y*, or it might deallocate *y* and then *x*.

10/2 {*AI95-00416-01*} {*freed: See nonexistent*} {*nonexistent*} {*exist (cease to)* [partial]} {*cease to exist (object)* [partial]} After Free(X), the object designated by X, and any subcomponents (and coextensions) thereof, no longer exist; their storage can be reused for other purposes.

Bounded (Run-Time) Errors

11 {*bounded error (cause)* [partial]} It is a bounded error to free a discriminated, unterminated task object. The possible consequences are:

11.a **Reason:** This is an error because the task might refer to its discriminants, and the discriminants might be deallocated by freeing the task object.

- 12 No exception is raised.
- 13 {*Program_Error (raised by failure of run-time check)*} {*Tasking_Error (raised by failure of run-time check)*} Program_Error or Tasking_Error is raised at the point of the deallocation.
- 14 {*Program_Error (raised by failure of run-time check)*} {*Tasking_Error (raised by failure of run-time check)*} Program_Error or Tasking_Error is raised in the task the next time it references any of the discriminants.

14.a **Implementation Note:** This last case presumes an implementation where the task references its discriminants indirectly, and the pointer is nulled out when the task object is deallocated.

15 In the first two cases, the storage for the discriminants (and for any enclosing object if it is designated by an access discriminant of the task) is not reclaimed prior to task termination.

15.a **Ramification:** The storage might never be reclaimed.

Erroneous Execution

16 {*nonexistent*} {*erroneous execution (cause)* [partial]} Evaluating a name that denotes a nonexistent object is erroneous. The execution of a call to an instance of Unchecked_Deallocation is erroneous if the object was created other than by an allocator for an access type whose pool is Name'Storage_Pool.

Implementation Advice

17 For a standard storage pool, Free should actually reclaim the storage.

17.a.1/2 **Implementation Advice:** For a standard storage pool, an instance of Unchecked_Deallocation should actually reclaim the storage.

17.a/2 **Ramification:** {*AI95-00114-01*} This is not a testable property, since we do not know how much storage is used by a given pool element, nor whether fragmentation can occur.

NOTES

18 27 The rules here that refer to Free apply to any instance of Unchecked_Deallocation.

19 28 Unchecked_Deallocation cannot be instantiated for an access-to-constant type. This is implied by the rules of 12.5.4.

Wording Changes from Ada 95

19.a/2 {*AI95-00416-01*} The rules for coextensions are clarified (mainly by adding that term). In theory, this reflects no change from Ada 95 (coextensions existed in Ada 95, they just didn't have a name).

13.11.3 Pragma Controlled

1 [Pragma Controlled is used to prevent any automatic reclamation of storage (garbage collection) for the objects created by allocators of a given access type.]

Syntax

2 The form of a pragma Controlled is as follows:

3 **pragma** Controlled(*first_subtype*_local_name);

3.a **Discussion:** Not to be confused with type Finalization.Controlled.

Legality Rules

4 The *first_subtype*_local_name of a pragma Controlled shall denote a non-derived access subtype.

Static Semantics

5 {*representation pragma (Controlled)* [partial]} {*pragma, representation (Controlled)* [partial]} A pragma Controlled is a representation pragma {*aspect of representation (controlled)* [partial]} {*controlled (aspect of representation)*} that specifies the *controlled* aspect of representation.

6 {*garbage collection*} *Garbage collection* is a process that automatically reclaims storage, or moves objects to a different address, while the objects still exist.

6.a **Ramification:** Storage reclamation upon leaving a master is not considered garbage collection.

6.b Note that garbage collection includes compaction of a pool ("moved to a different Address"), even if storage reclamation is not done.

6.c **Reason:** Programs that will be damaged by automatic storage reclamation are just as likely to be damaged by having objects moved to different locations in memory. A pragma Controlled should turn off both flavors of garbage collection.

6.d **Implementation Note:** If garbage collection reclaims the storage of a controlled object, it should first finalize it. Finalization is not done when moving an object; any self-relative pointers will have to be updated by the garbage collector. If an implementation provides garbage collection for a storage pool containing controlled objects (see 7.6), then it should provide a means for deferring garbage collection of those controlled objects.

6.e **Reason:** This allows the manager of a resource released by a Finalize operation to defer garbage collection during its critical regions; it is up to the author of the Finalize operation to do so. Garbage collection, at least in some systems, can happen asynchronously with respect to normal user code. Note that it is not enough to defer garbage collection during Initialize, Adjust, and Finalize, because the resource in question might be used in other situations as well. For example:

6.f
```
with Ada.Finalization;
package P is
```

6.g
```
    type My_Controlled is
        new Ada.Finalization.Limited_Controlled with private;
    procedure Finalize(Object : in out My_Controlled);
    type My_Controlled_Access is access My_Controlled;
```

6.h
```
    procedure Non_Reentrant;
```

6.i
```
private
    ...
end P;
```

6.j
```
package body P is
    X : Integer := 0;
    A : array(Integer range 1..10) of Integer;
```

6.k
```
        procedure Non_Reentrant is
        begin
            X := X + 1;
            -- If the system decides to do a garbage collection here,
            -- then we're in trouble, because it will call Finalize on
            -- the collected objects; we essentially have two threads
            -- of control erroneously accessing shared variables.
            -- The garbage collector behaves like a separate thread
            -- of control, even though the user hasn't declared
            -- any tasks.
            A(X) := ...;
        end Non_Reentrant;
```
6.l
```
        procedure Finalize(Object : in out My_Controlled) is
        begin
            Non_Reentrant;
        end Finalize;
    end P;
```
6.m
```
    with P; use P;
    procedure Main is
    begin
        ... new My_Controlled ... -- allocate some objects
        ...  forget the pointers to some of them, so they become garbage
        Non_Reentrant;
    end Main;
```

6.n It is the user's responsibility to protect against this sort of thing, and the implementation's responsibility to provide the necessary operations.

6.o We do not give these operations names, nor explain their exact semantics, because different implementations of garbage collection might have different needs, and because garbage collection is not supported by most Ada implementations, so portability is not important here. Another reason not to turn off garbage collection during each entire Finalize operation is that it would create a serial bottleneck; it might be only part of the Finalize operation that conflicts with some other resource. It is the intention that the mechanisms provided be finer-grained than pragma Controlled.

7 If a pragma Controlled is specified for an access type with a standard storage pool, then garbage collection is not performed for objects in that pool.

7.a **Ramification:** If Controlled is not specified, the implementation may, but need not, perform garbage collection. If Storage_Pool is specified, then a pragma Controlled for that type is ignored.

7.b **Reason:** Controlled means that implementation-provided garbage collection is turned off; if the Storage_Pool is specified, the pool controls whether garbage collection is done.

Implementation Permissions

8 An implementation need not support garbage collection, in which case, a pragma Controlled has no effect.

Wording Changes from Ada 83

8.a Ada 83 used the term “automatic storage reclamation” to refer to what is known traditionally as “garbage collection”. Because of the existence of storage pools (see 13.11), we need to distinguish this from the storage reclamation that might happen upon leaving a master. Therefore, we now use the term “garbage collection” in its normal computer-science sense. This has the additional advantage of making our terminology more accessible to people outside the Ada world.

13.12 Pragma Restrictions

1 [A pragma Restrictions expresses the user's intent to abide by certain restrictions. This may facilitate the construction of simpler run-time environments.]

Syntax

2 The form of a pragma Restrictions is as follows:

3 **pragma** Restrictions(restriction{, restriction});

4/2 {*AI95-00381-01*} restriction ::= *restriction_*identifier
  | *restriction_parameter_*identifier => restriction_parameter_argument

4.1/2 {*AI95-00381-01*} restriction_parameter_argument ::= name | expression

Name Resolution Rules

5 {*expected type (restriction parameter expression)* [partial]} Unless otherwise specified for a particular restriction, the expression is expected to be of any integer type.

Legality Rules

6 Unless otherwise specified for a particular restriction, the expression shall be static, and its value shall be nonnegative.

Static Semantics

7/2 {*AI95-00394-01*} The set of restrictions is implementation defined.

7.a/2 **Implementation defined:** The set of restrictions allowed in a pragma Restrictions.

Post-Compilation Rules

8 {*configuration pragma (Restrictions)* [partial]} {*pragma, configuration (Restrictions)* [partial]} A pragma Restrictions is a configuration pragma; unless otherwise specified for a particular restriction, a partition shall obey the restriction if a pragma Restrictions applies to any compilation unit included in the partition.

8.1/1 {*8652/0042*} {*AI95-00130-01*} For the purpose of checking whether a partition contains constructs that violate any restriction (unless specified otherwise for a particular restriction):

8.2/1 • {*8652/0042*} {*AI95-00130-01*} Generic instances are logically expanded at the point of instantiation;

8.3/1 • {*8652/0042*} {*AI95-00130-01*} If an object of a type is declared or allocated and not explicitly initialized, then all expressions appearing in the definition for the type and any of its ancestors are presumed to be used;

8.4/1 • {*8652/0042*} {*AI95-00130-01*} A default_expression for a formal parameter or a generic formal object is considered to be used if and only if the corresponding actual parameter is not provided in a given call or instantiation.

Implementation Permissions

9 An implementation may place limitations on the values of the expression that are supported, and limitations on the supported combinations of restrictions. The consequences of violating such limitations are implementation defined.

9.a **Implementation defined:** The consequences of violating limitations on Restrictions pragmas.

9.b **Ramification:** Such limitations may be enforced at compile time or at run time. Alternatively, the implementation is allowed to declare violations of the restrictions to be erroneous, and not enforce them at all.

9.1/1 {*8652/0042*} {*AI95-00130-01*} An implementation is permitted to omit restriction checks for code that is recognized at compile time to be unreachable and for which no code is generated.

9.2/1 {*8652/0043*} {*AI95-00190-01*} Whenever enforcement of a restriction is not required prior to execution, an implementation may nevertheless enforce the restriction prior to execution of a partition to which the restriction applies, provided that every execution of the partition would violate the restriction.

NOTES

10/2 29 {*AI95-00347-01*} Restrictions intended to facilitate the construction of efficient tasking run-time systems are defined in D.7. Restrictions intended for use when constructing high integrity systems are defined in H.4.

30 An implementation has to enforce the restrictions in cases where enforcement is required, even if it chooses not to take advantage of the restrictions in terms of efficiency. 11

Discussion: It is not the intent that an implementation will support a different run-time system for every possible combination of restrictions. An implementation might support only two run-time systems, and document a set of restrictions that is sufficient to allow use of the more efficient and safe one. 11.a

Extensions to Ada 83

{*extensions to Ada 83*} Pragma Restrictions is new to Ada 95. 11.b

Wording Changes from Ada 95

{*8652/0042*} {*AI95-00130-01*} **Corrigendum:** Corrected the wording so that restrictions are checked inside of generic instantiations and in default expressions. Since not making these checks would violate the purpose of restrictions, we are not documenting this as an incompatibility. 11.c/2

{*8652/0043*} {*AI95-00190-01*} **Corrigendum:** Added a permission that restrictions can be enforced at compile-time. While this is technically incompatible, documenting it as such would be unnecessarily alarming - there should not be any programs depending on the runtime failure of restrictions. 11.d/2

{*AI95-00381-01*} The syntax of a restriction_parameter_argument has been defined to better support restriction No_Dependence (see 13.12.1). 11.e/2

13.12.1 Language-Defined Restrictions

Static Semantics

{*AI95-00257-01*} The following *restriction_*identifiers are language-defined (additional restrictions are defined in the Specialized Needs Annexes): 1/2

{*AI95-00257-01*} {*Restrictions (No_Implementation_Attributes)*} No_Implementation_Attributes 2/2
There are no implementation-defined attributes. This restriction applies only to the current compilation or environment, not the entire partition.

Discussion: This restriction (as well as No_Implementation_Pragmas) only applies to the current compilation, because it is likely that the runtime (and possibly user-written low-level code) will need to use implementation-defined entities. But a partition-wide restriction applies everywhere, including the runtime. 2.a/2

{*AI95-00257-01*} {*Restrictions (No_Implementation_Pragmas)*} No_Implementation_Pragmas 3/2
There are no implementation-defined pragmas or pragma arguments. This restriction applies only to the current compilation or environment, not the entire partition.

{*AI95-00368-01*} {*Restrictions (No_Obsolescent_Features)*} No_Obsolescent_Features 4/2
There is no use of language features defined in Annex J. It is implementation-defined if uses of the renamings of J.1 are detected by this restriction. This restriction applies only to the current compilation or environment, not the entire partition.

Reason: A user could compile a rename like 4.a/2

4.b/2
```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```

Such a rename must not be disallowed by this restriction, nor should the compilation of such a rename be restricted by an implementation. Many implementations implement the renames of J.1 by compiling them normally; we do not want to require implementations to use a special mechanism to implement these renames. 4.c/2

{*AI95-00381-01*} The following *restriction_parameter_*identifier is language defined: 5/2

{*AI95-00381-01*} {*Restrictions (No_Dependence)*} No_Dependence 6/2
Specifies a library unit on which there are no semantic dependences.

Legality Rules

{*AI95-00381-01*} The restriction_parameter_argument of a No_Dependence restriction shall be a name; the name shall have the form of a full expanded name of a library unit, but need not denote a unit present in the environment. 7/2

7.a/2 **Ramification:** This name is not resolved.

Post-Compilation Rules

8/2 {*AI95-00381-01*} No compilation unit included in the partition shall depend semantically on the library unit identified by the name.

8.a/2 **Ramification:** There is no requirement that the library unit actually exist. One possible use of the pragma is to prevent the use of implementation-defined units; when the program is ported to a different compiler, it is perfectly reasonable that no unit with the name exist.

Extensions to Ada 95

8.b/2 {*AI95-00257-01*} {*AI95-00368-01*} {*extensions to Ada 95*} Restrictions No_Implementation_Attributes, No_Implementation_Pragmas, and No_Obsolescent_Features are new.

8.c/2 {*AI95-00381-01*} Restriction No_Dependence is new.

13.13 Streams

1 {*stream*} {*stream type*} A *stream* is a sequence of elements comprising values from possibly different types and allowing sequential access to these values. A *stream type* is a type in the class whose root type is Streams.Root_Stream_Type. A stream type may be implemented in various ways, such as an external sequential file, an internal buffer, or a network channel.

1.a **Discussion:** A stream element will often be the same size as a storage element, but that is not required.

Extensions to Ada 83

1.b {*extensions to Ada 83*} Streams are new in Ada 95.

13.13.1 The Package Streams

Static Semantics

1 The abstract type Root_Stream_Type is the root type of the class of stream types. The types in this class represent different kinds of streams. A new stream type is defined by extending the root type (or some other stream type), overriding the Read and Write operations, and optionally defining additional primitive subprograms, according to the requirements of the particular kind of stream. The predefined stream-oriented attributes like T'Read and T'Write make dispatching calls on the Read and Write procedures of the Root_Stream_Type. (User-defined T'Read and T'Write attributes can also make such calls, or can call the Read and Write attributes of other types.)

2
```
package Ada.Streams is
    pragma Pure(Streams){unpolluted} ;
```

3/2
```
{AI95-00161-01}     type Root_Stream_Type is abstract tagged limited private;
    pragma Preelaborable_Initialization(Root_Stream_Type);
```

4/1
```
{8652/0044} {AI95-00181-01}     type Stream_Element is mod implementation-defined;
    type Stream_Element_Offset is range implementation-defined;
    subtype Stream_Element_Count is
        Stream_Element_Offset range 0..Stream_Element_Offset'Last;
    type Stream_Element_Array is
        array(Stream_Element_Offset range <>) of aliased Stream_Element;
```

5
```
    procedure Read(
      Stream : in out Root_Stream_Type;
      Item   : out Stream_Element_Array;
      Last   : out Stream_Element_Offset) is abstract;
```

6
```
    procedure Write(
      Stream : in out Root_Stream_Type;
      Item   : in Stream_Element_Array) is abstract;
```

```
private                                                                 7
   ... -- not specified by the language
end Ada.Streams;
```

{*AI95-00227-01*} The Read operation transfers stream elements from the specified stream to fill the array Item. Elements are transferred until Item'Length elements have been transferred, or until the end of the stream is reached. If any elements are transferred, the index of the last stream element transferred is returned in Last. Otherwise, Item'First - 1 is returned in Last. Last is less than Item'Last only if the end of the stream is reached. 8/2

The Write operation appends Item to the specified stream. 9

Discussion: {*AI95-00114-01*} The index subtype of Stream_Element_Array is Stream_Element_Offset because we wish to allow maximum flexibility. Most Stream_Element_Arrays will probably have a lower bound of 0 or 1, but other lower bounds, including negative ones, make sense in some situations. 9.a/2

{*AI95-00114-01*} Note that there are some language-defined subprograms that fill part of a Stream_Element_Array, and return the index of the last element filled as a Stream_Element_Offset. The Read procedures declared here, Streams.Stream_IO (see A.12.1), and System.RPC (see E.5) behave in this manner. These will raise Constraint_Error if the resulting Last value is not in Stream_Element_Offset. This implies that the Stream_Element_Array passed to these subprograms should not have a lower bound of Stream_Element_Offset'First, because then a read of 0 elements would always raise Constraint_Error. A better choice of lower bound is 1. 9.b/2

Implementation Permissions

{*8652/0044*} {*AI95-00181-01*} If Stream_Element'Size is not a multiple of System.Storage_Unit, then the components of Stream_Element_Array need not be aliased. 9.1/1

Ramification: {*AI95-00114-01*} If the Stream_Element'Size is less than the size of System.Storage_Unit, then components of Stream_Element_Array need not be aliased. This is necessary as the components of type Stream_Element size might not be addressable on the target architecture. 9.b.1/2

NOTES

31 See A.12.1, “The Package Streams.Stream_IO” for an example of extending type Root_Stream_Type. 10

32 {*AI95-00227-01*} If the end of stream has been reached, and Item'First is Stream_Element_Offset'First, Read will raise Constraint_Error. 11/2

Ramification: Thus, Stream_Element_Arrays should start at 0 or 1, not Stream_Element_Offset'First. 11.a/2

Extensions to Ada 95

{*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Added pragma Preelaborable_Initialization to type Root_Stream_Type. 11.b/2

Wording Changes from Ada 95

{*8652/0044*} {*AI95-00181-01*} **Corrigendum:** Stream elements are aliased presuming that makes sense. 11.c/2

{*AI95-00227-01*} Fixed the wording for Read to properly define the result in Last when no stream elements are transfered. 11.d/2

13.13.2 Stream-Oriented Attributes

{*8652/0009*} {*AI95-00137-01*} The operational attributes Write, Read, Output, and Input convert values to a stream of elements and reconstruct values from a stream. 1/1

Static Semantics

{*AI95-00270-01*} For every subtype S of an elementary type *T*, the following representation attribute is defined: 1.1/2

S'Stream_Size 1.2/2

{*AI95-00270-01*} Denotes the number of bits occupied in a stream by items of subtype S. Hence, the number of stream elements required per item of elementary type *T* is:

```
T'Stream_Size / Ada.Streams.Stream_Element'Size                         1.3/2
```

1.4/2 The value of this attribute is of type *universal_integer* and is a multiple of Stream_Element'Size.

1.5/2 Stream_Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, nonnegative, and a multiple of Stream_Element'Size.

1.a/2 **Discussion:** Stream_Size is a type-related attribute (see 13.1).

Implementation Advice

1.6/2 {*AI95-00270-01*} If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size.

1.b/2 **Implementation Advice:** If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size.

1.c/2 **Reason:** {*AI95-00270-01*} This is Implementation Advice because we want to allow implementations to remain compatible with their Ada 95 implementations, which may have a different handling of the number of stream elements. Users can always specify Stream_Size if they need a specific number of stream elements.

1.7/2 {*AI95-00270-01*} {*recommended level of support (Stream_Size attribute)* [partial]} The recommended level of support for the Stream_Size attribute is:

1.8/2
- {*AI95-00270-01*} A Stream_Size clause should be supported for a discrete or fixed point type *T* if the specified Stream_Size is a multiple of Stream_Element'Size and is no less than the size of the first subtype of *T*, and no greater than the size of the largest type of the same elementary class (signed integer, modular integer, enumeration, ordinary fixed point, or decimal fixed point).

1.d/2 **Implementation Advice:** The recommended level of support for the Stream_Size attribute should be followed.

1.e/2 **Ramification:** There are no requirements beyond supporting confirming Stream_Size clauses for floating point and access types. Floating point and access types usually only have a handful of defined formats, streaming anything else makes no sense for them.

1.f/2 For discrete and fixed point types, this may require support for sizes other than the “natural” ones. For instance, on a typical machine with 32-bit integers and a Stream_Element'Size of 8, setting Stream_Size to 24 must be supported. This is required as such formats can be useful for interoperability with unusual machines, and there is no difficulty with the implementation (drop extra bits on output, sign extend on input).

Static Semantics

2 For every subtype S of a specific type *T*, the following attributes are defined.

3 S'Write S'Write denotes a procedure with the following specification:

4/2
```
{AI95-00441-01} procedure S'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```

5 S'Write writes the value of *Item* to *Stream*.

6 S'Read S'Read denotes a procedure with the following specification:

7/2
```
{AI95-00441-01} procedure S'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T)
```

8 S'Read reads the value of *Item* from *Stream*.

8.1/2 {*8652/0040*} {*AI95-00108-01*} {*AI95-00444-01*} For an untagged derived type, the Write (resp. Read) attribute is inherited according to the rules given in 13.1 if the attribute is available for the parent type at the point where *T* is declared. For a tagged derived type, these attributes are not inherited, but rather the default implementations are used.

{*AI95-00444-01*} The default implementations of the Write and Read attributes, where available, execute as follows: 8.2/2

{*8652/0040*} {*AI95-00108-01*} {*AI95-00195-01*} {*AI95-00251-01*} {*AI95-00270-01*} For elementary types, Read reads (and Write writes) the number of stream elements implied by the Stream_Size for the type *T*; the representation of those stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in canonical order, which is last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included. For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of the parent type or any progenitor type of *T* is available anywhere within the immediate scope of *T*, and the attribute of the parent type or the type of any of the extension components is not available at the freezing point of *T*, then the attribute of *T* shall be directly specified. 9/2

Implementation defined: The contents of the stream elements read and written by the Read and Write attributes of elementary types. 9.a/2

Reason: A discriminant with a default value is treated simply as a component of the object. On the other hand, an array bound or a discriminant without a default value, is treated as “descriptor” or “dope” that must be provided in order to create the object and thus is logically separate from the regular components. Such “descriptor” data are written by 'Output and produced as part of the delivered result by the 'Input function, but they are not written by 'Write nor read by 'Read. A tag is like a discriminant without a default. 9.b

{*8652/0040*} {*AI95-00108-01*} For limited type extensions, we must have a definition of 'Read and 'Write if the parent type has one, as it is possible to make a dispatching call through the attributes. The rule is designed to automatically do the right thing in as many cases as possible. 9.b.1/1

{*AI95-00251-01*} Similarly, a type that has a progenitor with an available attribute must also have that attribute, for the same reason. 9.b.2/1

Ramification: {*AI95-00195-01*} For a composite object, the subprogram denoted by the Write or Read attribute of each component is called, whether it is the default or is user-specified. Implementations are allowed to optimize these calls (see below), presuming the properties of the attributes are preserved. 9.c/2

{*AI95-00270-01*} Constraint_Error is raised by the predefined Write attribute if the value of the elementary item is outside the range of values representable using Stream_Size bits. For a signed integer type, an enumeration type, or a fixed point type, the range is unsigned only if the integer code for the lower bound of the first subtype is nonnegative, and a (symmetric) signed range that covers all values of the first subtype would require more than Stream_Size bits; otherwise the range is signed. 9.1/2

For every subtype S'Class of a class-wide type *T*'Class: 10

S'Class'Write 11

S'Class'Write denotes a procedure with the following specification:

12/2
```
{AI95-00441-01} procedure S'Class'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item. 13

S'Class'Read 14

S'Class'Read denotes a procedure with the following specification:

15/2
```
{AI95-00441-01} procedure S'Class'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T'Class)
```

Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item. 16

16.a **Reason:** It is necessary to have class-wide versions of Read and Write in order to avoid generic contract model violations; in a generic, we don't necessarily know at compile time whether a given type is specific or class-wide.

Implementation Advice

17/2 *This paragraph was deleted.*{*AI95-00270-01*}

Static Semantics

18 For every subtype S of a specific type *T*, the following attributes are defined.

19 S'Output S'Output denotes a procedure with the following specification:

20/2
```
{AI95-00441-01} procedure S'Output(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```

21 S'Output writes the value of *Item* to *Stream*, including any bounds or discriminants.

21.a **Ramification:** Note that the bounds are included even for an array type whose first subtype is constrained.

22 S'Input S'Input denotes a function with the following specification:

23/2
```
{AI95-00441-01} function S'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T
```

24 S'Input reads and returns one value from *Stream*, using any bounds or discriminants written by a corresponding S'Output to determine how much to read.

25/2 {*8652/0040*} {*AI95-00108-01*} {*AI95-00444-01*} For an untagged derived type, the Output (resp. Input) attribute is inherited according to the rules given in 13.1 if the attribute is available for the parent type at the point where *T* is declared. For a tagged derived type, these attributes are not inherited, but rather the default implementations are used.

25.1/2 {*AI95-00444-01*} The default implementations of the Output and Input attributes, where available, execute as follows:

26
- If *T* is an array type, S'Output first writes the bounds, and S'Input first reads the bounds. If *T* has discriminants without defaults, S'Output first writes the discriminants (using S'Write for each), and S'Input first reads the discriminants (using S'Read for each).

27/2
- {*AI95-00195-01*} S'Output then calls S'Write to write the value of *Item* to the stream. S'Input then creates an object (with the bounds or discriminants, if any, taken from the stream), passes it to S'Read, and returns the value of the object. Normal default initialization and finalization take place for this object (see 3.3.1, 7.6, and 7.6.1).

27.1/2 {*AI95-00251-01*} If *T* is an abstract type, then S'Input is an abstract function.

27.a/2 **Ramification:** For an abstract type *T*, S'Input can be called in a dispatching call, or passed to a abstract formal subprogram. But it cannot be used in non-dispatching contexts, because we don't allow objects of abstract types to exist. The designation of this function as abstract has no impact on descendants of *T*, as *T*'Input is not inherited for tagged types, but rather recreated (and the default implementation of *T*'Input calls *T*'Read, not the parent type's *T*'Input). Note that *T*'Input cannot be specified in this case, as any function with the proper profile is necessarily abstract, and specifying abstract subprograms in an attribute_definition_clause is illegal.

28 For every subtype S'Class of a class-wide type *T*'Class:

29 S'Class'Output

S'Class'Output denotes a procedure with the following specification:

30/2
```
{AI95-00441-01} procedure S'Class'Output(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

31/2 {*AI95-00344-01*} First writes the external tag of *Item* to *Stream* (by calling String'Output(*Stream*, Tags.External_Tag(*Item*'Tag)) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag.

Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S.

31.a/2 **Reason:** {*AI95-00344-01*} We raise Tag_Error here for nested types as such a type cannot be successfully read with S'Class'Input, and it doesn't make sense to allow writing a value that cannot be read.

32 S'Class'Input

S'Class'Input denotes a function with the following specification:

33/2
```
{AI95-00441-01} function S'Class'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T'Class
```

34/2 {*AI95-00279-01*} {*AI95-00344-01*} First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which might raise Tag_Error — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. If the specific type identified by the internal tag is not covered by *T*'Class or is abstract, Constraint_Error is raised.

35/2 {*AI95-00195-01*} {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} In the default implementation of Read and Input for a composite type, for each scalar component that is a discriminant or whose component_declaration includes a default_expression, a check is made that the value returned by Read for the component belongs to its subtype. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails. For other scalar components, no check is made. For each component that is of an access type, if the implementation can detect that the value returned by Read for the component is not a value of its subtype, Constraint_Error is raised. If the value is not a value of its subtype and this error is not detected, the component has an abnormal value, and erroneous execution can result (see 13.9.1). In the default implementation of Read for a composite type with defaulted discriminants, if the actual parameter of Read is constrained, a check is made that the discriminants read from the stream are equal to those of the actual parameter. Constraint_Error is raised if this check fails.

36/2 {*AI95-00195-01*} {*unspecified* [partial]} It is unspecified at which point and in which order these checks are performed. In particular, if Constraint_Error is raised due to the failure of one of these checks, it is unspecified how many stream elements have been read from the stream.

37/1 {*8652/0045*} {*AI95-00132-01*} {*End_Error (raised by failure of run-time check)*} In the default implementation of Read and Input for a type, End_Error is raised if the end of the stream is reached before the reading of a value of the type is completed.

38/2 {*8652/0040*} {*AI95-00108-01*} {*AI95-00195-01*} {*AI95-00251-01*} {*specifiable (of Read for a type)* [partial]} {*specifiable (of Write for a type)* [partial]} {*specifiable (of Input for a type)* [partial]} {*specifiable (of Output for a type)* [partial]} {*Read clause*} {*Write clause*} {*Input clause*} {*Output clause*} The stream-oriented attributes may be specified for any type via an attribute_definition_clause. The subprogram name given in such a clause shall not denote an abstract subprogram. Furthermore, if a stream-oriented attribute is specified for an interface type by an attribute_definition_clause, the subprogram name given in the clause shall statically denote a null procedure.

38.a/2 *This paragraph was deleted.*{*AI95-00195-01*}

38.a.1/2 *This paragraph was deleted.*{*8652/0040*} {*AI95-00108-01*} {*AI95-00195-01*}

38.b/2 **Discussion:** {*AI95-00251-01*} Stream attributes (other than Input) are always null procedures for interface types (they have no components). We need to allow explicit setting of the Read and Write attributes in order that the class-wide attributes like LI'Class'Input can be made available. (In that case, any descendant of the interface type would require available attributes.) But we don't allow any concrete implementation because these don't participate in extensions (unless the interface is the parent type). If we didn't ban concrete implementations, the order of declaration of a pair of interfaces would become significant. For example, if Int1 and Int2 are interfaces with concrete implementations of 'Read, then the following declarations would have different implementations for 'Read:

38.c/2
```
        type Con1 is new Int1 and Int2 with null record;
        type Con2 is new Int2 and Int1 with null record;
```

38.d/2 This would violate our design principle that the order of the specification of the interfaces in a derived_type_definition doesn't matter.

38.e/2 **Ramification:** The Input attribute cannot be specified for an interface. As it is a function, a null procedure is impossible; a concrete function is not possible anyway as any function returning an abstract type must be abstract. And we don't allow specifying stream attributes to be abstract subprograms. This has no impact, as the availability of Int'Class'Input (where Int is a limited interface) depends on whether Int'Read (not Int'Input) is specified. There is no reason to allow Int'Output to be specified, either, but there is equally no reason to disallow it, so we don't have a special rule for that.

38.f/2 **Discussion:** {*AI95-00195-01*} Limited types generally do not have default implementations of the stream-oriented attributes. The rules defining when a stream-oriented attribute is available (see below) determine when an attribute of a limited type is in fact well defined and usable. The rules are designed to maximize the number of cases in which the attributes are usable. For instance, when the language provides a default implementation of an attribute for a limited type based on a specified attribute for the parent type, we want to be able to call that attribute.

39/2 {*AI95-00195-01*} A stream-oriented attribute for a subtype of a specific type *T* is *available* at places where one of the following conditions is true: {*available (stream attribute)*}

40/2 • *T* is nonlimited.

41/2 • The attribute_designator is Read (resp. Write) and *T* is a limited record extension, and the attribute Read (resp. Write) is available for the parent type of *T* and for the types of all of the extension components.

41.a/2 **Reason:** In this case, the language provides a well-defined default implementation, which we want to be able to call.

42/2 • *T* is a limited untagged derived type, and the attribute was inherited for the type.

42.a/2 **Reason:** Attributes are only inherited for untagged derived types, and surely we want to be able to call inherited attributes.

43/2 • The attribute_designator is Input (resp. Output), and *T* is a limited type, and the attribute Read (resp. Write) is available for *T*.

43.a/2 **Reason:** The default implementation of Input and Output are based on Read and Write; so if the implementation of Read or Write is good, so is the matching implementation of Input or Output.

44/2 • The attribute has been specified via an attribute_definition_clause, and the attribute_definition_clause is visible.

44.a/2 **Reason:** We always want to allow calling a specified attribute. But we don't want availability to break privacy. Therefore, only attributes whose specification can be seen count. Yes, we defined the visibility of an attribute_definition_clause (see 8.3).

45/2 {*AI95-00195-01*} A stream-oriented attribute for a subtype of a class-wide type *T*'Class is available at places where one of the following conditions is true:

46/2 • *T* is nonlimited;

47/2 • the attribute has been specified via an attribute_definition_clause, and the attribute_definition_clause is visible; or

48/2 • the corresponding attribute of *T* is available, provided that if *T* has a partial view, the corresponding attribute is available at the end of the visible part where *T* is declared.

48.a/2 **Reason:** The rules are stricter for class-wide attributes because (for the default implementation) we must ensure that any specific attribute that might ever be dispatched to is available. Because we require specification of attributes for extensions of limited parent types with available attributes, we can in fact know this. Otherwise, we would not be able to use default class-wide attributes with limited types, a significant limitation.

49/2 {*AI95-00195-01*} An attribute_reference for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the attribute_reference. Furthermore, an attribute_reference for *T*'Input is illegal if *T* is an abstract type.

Discussion: Stream attributes always exist. It is illegal to call them in some cases. Having the attributes not be defined for some limited types would seem to be a cleaner solution, but it would lead to contract model problems for limited private types. 49.a/2

T'Input is available for abstract types so that *T*'Class'Input is available. But we certainly don't want to allow calls that could create an object of an abstract type. Remember that *T*'Class is never abstract, so the above legality rule doesn't apply to it. We don't have to discuss whether the attribute is specified, as it cannot be: any function returning the type would have to be abstract, and we do not allow specifying an attribute with an abstract subprogram. 49.b/2

{*AI95-00195-01*} In the parameter_and_result_profiles for the stream-oriented attributes, the subtype of the Item parameter is the base subtype of *T* if *T* is a scalar type, and the first subtype otherwise. The same rule applies to the result of the Input attribute. 50/2

{*AI95-00195-01*} For an attribute_definition_clause specifying one of these attributes, the subtype of the Item parameter shall be the base subtype if scalar, and the first subtype otherwise. The same rule applies to the result of the Input function. 51/2

Reason: This is to simplify implementation. 51.a/2

Ramification: The view of the type at the point of the attribute_definition_clause determines whether the first subtype or base subtype is required. Thus, for a scalar type with a partial view (which is never scalar), whether the first subtype or the base subtype is required is determined by whether the attribute_definition_clause occurs before or after the full definition of the scalar type. 51.b/2

{*AI95-00366-01*} {*support external streaming*} {*external streaming (type supports)*} [A type is said to *support external streaming* if Read and Write attributes are provided for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling the representation.] A limited type supports external streaming only if it has available Read and Write attributes. A type with a part that is of an access type supports external streaming only if that access type or the type of some part that includes the access type component, has Read and Write attributes that have been specified via an attribute_definition_clause, and that attribute_definition_clause is visible. [An anonymous access type does not support external streaming.]All other types support external streaming. 52/2

Ramification: A limited type with a part that is of an access type needs to satisfy both rules. 52.a/2

Erroneous Execution

{*AI95-00279-01*} {*AI95-00344-01*} {*erroneous execution (cause)* [partial]} If the internal tag returned by Descendant_Tag to T'Class'Input identifies a type that is not library-level and whose tag has not been created, or does not exist in the partition at the time of the call, execution is erroneous. 53/2

Ramification: The definition of Descendant_Tag prevents such a tag from being provided to T'Class'Input if T is a library-level type. However, this rule is needed for nested tagged types. 53.a/2

Implementation Requirements

{*8652/0040*} {*AI95-00108-01*} For every subtype *S* of a language-defined nonlimited specific type *T*, the output generated by S'Output or S'Write shall be readable by S'Input or S'Read, respectively. This rule applies across partitions if the implementation conforms to the Distributed Systems Annex. 54/1

{*AI95-00195-01*} If Constraint_Error is raised during a call to Read because of failure of one the above checks, the implementation must ensure that the discriminants of the actual parameter of Read are not modified. 55/2

Implementation Permissions

{*AI95-00195-01*} The number of calls performed by the predefined implementation of the stream-oriented attributes on the Read and Write operations of the stream type is unspecified. An implementation may take advantage of this permission to perform internal buffering. However, all the calls on the Read and Write operations of the stream type needed to implement an explicit invocation of a stream-oriented 56/2 attribute must take place before this invocation returns. An explicit invocation is one appearing explicitly in the program text, possibly through a generic instantiation (see 12.3).

NOTES

57 33 For a definite subtype S of a type *T*, only *T*'Write and *T*'Read are needed to pass an arbitrary value of the subtype through a stream. For an indefinite subtype S of a type *T*, *T*'Output and *T*'Input will normally be needed, since *T*'Write and *T*'Read do not pass bounds, discriminants, or tags.

58 34 User-specified attributes of S'Class are not inherited by other class-wide types descended from S.

Examples

59 *Example of user-defined Write attribute:*

60/2
```
{AI95-00441-01} procedure My_Write(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item   : My_Integer'Base);
for My_Integer'Write use My_Write;
```

60.a **Discussion:** *Example of network input/output using input output attributes:*

60.b
```
with Ada.Streams; use Ada.Streams;
generic
    type Msg_Type(<>) is private;
package Network_IO is
    -- Connect/Disconnect are used to establish the stream
    procedure Connect(...);
    procedure Disconnect(...);
```

60.c
```
    -- Send/Receive transfer messages across the network
    procedure Send(X : in Msg_Type);
    function Receive return Msg_Type;
private
    type Network_Stream is new Root_Stream_Type with ...
    procedure Read(...);  -- define Read/Write for Network_Stream
    procedure Write(...);
end Network_IO;
```

60.d
```
with Ada.Streams; use Ada.Streams;
package body Network_IO is
    Current_Stream : aliased Network_Stream;
    . . .
    procedure Connect(...) is ...;
    procedure Disconnect(...) is ...;
```

60.e
```
    procedure Send(X : in Msg_Type) is
    begin
        Msg_Type'Output(Current_Stream'Access, X);
    end Send;
```

60.f
```
    function Receive return Msg_Type is
    begin
        return Msg_Type'Input(Current_Stream'Access);
    end Receive;
end Network_IO;
```

Inconsistencies With Ada 95

60.g/2 {*8652/0040*} {*AI95-00108-01*} {*inconsistencies with Ada 95*} **Corrigendum:** Clarified how the default implementation for stream attributes is determined (eliminating conflicting language). The new wording provides that attributes for type extensions are created by composing the parent's attribute with those for the extension components if any. If a program was written assuming that the extension components were not included in the stream (as in original Ada 95), it would fail to work in the language as corrected by the Corrigendum.

60.h/2 {*AI95-00195-01*} **Amendment Correction:** Explicitly provided a permission that the number of calls to the underlying stream Read and Write operations may differ from the number determined by the canonical operations. If Ada 95 code somehow depended on the number of calls to Read or Write, it could fail with an Ada 2005 implementation. Such code is likely to be very rare; moreover, such code is really wrong, as the permission applies to Ada 95 as well.

Extensions to Ada 95

60.i/2 {*AI95-00270-01*} {*extensions to Ada 95*} The Stream_Size attribute is new. It allows specifying the number of bits that will be streamed for a type. The Implementation Advice involving this also was changed; this is not incompatible because Implementation Advice does not have to be followed.

{*8652/0040*} {*AI95-00108-01*} {*AI95-00195-01*} {*AI95-00444-01*} **Corrigendum:** Limited types may have default constructed attributes if all of the parent and (for extensions) extension components have available attributes. Ada 2005 adds the notion of availability to patch up some holes in the Corrigendum model. 60.j/2

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added wording to specify that these are operational attributes. 60.k/2

{*8652/0045*} {*AI95-00132-01*} **Corrigendum:** Clarified that End_Error is raised by the default implementation of Read and Input if the end of the stream is reached. (The result could have been abnormal without this clarification, thus this is not an inconsistency, as the programmer could not have depended on the previous behavior.) 60.l/2

{*AI95-00195-01*} Clarified that the default implementation of S'Input does normal initialization on the object that it passes to S'Read. 60.m/2

{*AI95-00195-01*} Explicitly stated that what is read from a stream when a required check fails is unspecified. 60.n/2

{*AI95-00251-01*} Defined availability and default implementations for types with progenitors. 60.o/2

{*AI95-00279-01*} Specified that Constraint_Error is raised if the internal tag retrieved for S'Class'Input is for some type not covered by S'Class or is abstract. We also explicitly state that the program is erroneous if the tag has not been created or does not currently exist in the partition. (Ada 95 did not specify what happened in these cases; it's very unlikely to have provided some useful result, so this is not considered an inconsistency.) 60.p/2

{*AI95-00344-01*} Added wording to support nested type extensions. S'Input and S'Output always raise Tag_Error for such extensions, and such extensions were not permitted in Ada 95, so this is neither an extension nor an incompatibility. 60.q/2

{*AI95-00366-01*} Defined *supports external streaming* to put all of the rules about “good” stream attributes in one place. This is used for distribution and for defining pragma Pure. 60.r/2

{*AI95-00441-01*} Added the **not null** qualifier to the first parameter of all of the stream attributes, so that the semantics doesn't change between Ada 95 and Ada 2005. This change is compatible, because mode conformance is required for subprograms specified as stream attributes, and null_exclusions are not considered for mode conformance. 60.s/2

{*AI95-00444-01*} Improved the wording to make it clear that we don't define the default implementations of attributes that cannot be called (that is, aren't “available”). Also clarified when inheritance takes place. 60.t/2

13.14 Freezing Rules

[This clause defines a place in the program text where each declared entity becomes “frozen.” A use of an entity, such as a reference to it by name, or (for a type) an expression of the type, causes freezing of the entity in some contexts, as described below. The Legality Rules forbid certain kinds of uses of an entity in the region of text where it is frozen.] 1

Reason: This concept has two purposes: a compile-time one and a run-time one. 1.a

The compile-time purpose of the freezing rules comes from the fact that the evaluation of static expressions depends on overload resolution, and overload resolution sometimes depends on the value of a static expression. (The dependence of static evaluation upon overload resolution is obvious. The dependence in the other direction is more subtle. There are three rules that require static expressions in contexts that can appear in declarative places: The expression in an attribute_designator shall be static. In a record aggregate, variant-controlling discriminants shall be static. In an array aggregate with more than one named association, the choices shall be static. The compiler needs to know the value of these expressions in order to perform overload resolution and legality checking.) We wish to allow a compiler to evaluate static expressions when it sees them in a single pass over the compilation_unit. The freezing rules ensure that. 1.b

The run-time purpose of the freezing rules is called the “linear elaboration model.” This means that declarations are elaborated in the order in which they appear in the program text, and later elaborations can depend on the results of earlier ones. The elaboration of the declarations of certain entities requires run-time information about the implementation details of other entities. The freezing rules ensure that this information has been calculated by the time it is used. For example, suppose the initial value of a constant is the result of a function call that takes a parameter of type *T*. In order to pass that parameter, the size of type *T* has to be known. If *T* is composite, that size might be known only at run time. 1.c

(Note that in these discussions, words like “before” and “after” generally refer to places in the program text, as opposed to times at run time.) 1.d

Discussion: The “implementation details” we're talking about above are: 1.e

1.f
- For a tagged type, the implementations of all the primitive subprograms of the type — that is (in the canonical implementation model), the contents of the type descriptor, which contains pointers to the code for each primitive subprogram.

1.g
- For a type, the full type declaration of any parts (including the type itself) that are private.

1.h
- For a deferred constant, the full constant declaration, which gives the constant's value. (Since this information necessarily comes after the constant's type and subtype are fully known, there's no need to worry about its type or subtype.)

1.i
- For any entity, representation information specified by the user via representation items. Most representation items are for types or subtypes; however, various other kinds of entities, such as objects and subprograms, are possible.

1.j Similar issues arise for incomplete types. However, we do not use freezing there; incomplete types have different, more severe, restrictions. Similar issues also arise for subprograms, protected operations, tasks and generic units. However, we do not use freezing there either; 3.11 prevents problems with run-time Elaboration_Checks.

Language Design Principles

1.k An evaluable construct should freeze anything that's needed to evaluate it.

1.l However, if the construct is not evaluated where it appears, let it cause freezing later, when it is evaluated. This is the case for default_expressions and default_names. (Formal parameters, generic formal parameters, and components can have default_expressions or default_names.)

1.m The compiler should be allowed to evaluate static expressions without knowledge of their context. (I.e. there should not be any special rules for static expressions that happen to occur in a context that requires a static expression.)

1.n Compilers should be allowed to evaluate static expressions (and record the results) using the run-time representation of the type. For example, suppose Color'Pos(Red) = 1, but the internal code for Red is 37. If the value of a static expression is Red, some compilers might store 1 in their symbol table, and other compilers might store 37. Either compiler design should be feasible.

1.o Compilers should never be required to detect erroneousness or exceptions at compile time (although it's very nice if they do). This implies that we should not require code-generation for a nonstatic expression of type *T* too early, even if we can prove that that expression will be erroneous, or will raise an exception.

1.p Here's an example (modified from AI83-00039, Example 3):

1.q
```
type T is
    record
        ...
    end record;
function F return T;
function G(X : T) return Boolean;
Y : Boolean := G(F); -- doesn't force T in Ada 83
for T use
    record
        ...
    end record;
```

1.r AI83-00039 says this is legal. Of course, it raises Program_Error because the function bodies aren't elaborated yet. A one-pass compiler has to generate code for an expression of type T before it knows the representation of T. Here's a similar example, which AI83-00039 also says is legal:

1.s
```
package P is
    type T is private;
    function F return T;
    function G(X : T) return Boolean;
    Y : Boolean := G(F); -- doesn't force T in Ada 83
private
    type T is
        record
            ...
        end record;
end P;
```

1.t If T's size were dynamic, that size would be stored in some compiler-generated dope; this dope would be initialized at the place of the full type declaration. However, the generated code for the function calls would most likely allocate a temp of the size specified by the dope *before* checking for Program_Error. That dope would contain uninitialized junk, resulting in disaster. To avoid doing that, the compiler would have to determine, at compile time, that the expression will raise Program_Error.

1.u This is silly. If we're going to require compilers to detect the exception at compile time, we might as well formulate the rule as a legality rule.

Compilers should not be required to generate code to load the value of a variable before the address of the variable has been determined. 1.v

After an entity has been frozen, no further requirements may be placed on its representation (such as by a representation item or a full_type_declaration). 1.w

{*freezing (entity)* [distributed]} {*freezing points (entity)*} The *freezing* of an entity occurs at one or more places (*freezing points*) in the program text where the representation for the entity has to be fully determined. Each entity is frozen from its first freezing point to the end of the program text (given the ordering of compilation units defined in 10.1.4). 2

Ramification: The "representation" for a subprogram includes its calling convention and means for referencing the subprogram body, either a "link-name" or specified address. It does not include the code for the subprogram body itself, nor its address if a link-name is used to reference the body. 2.a

{*8652/0014*} {*freezing (entity caused by the end of an enclosing construct)*} The end of a declarative_part, protected_body, or a declaration of a library package or generic library package, causes *freezing* of each entity declared within it, except for incomplete types. {*freezing (entity caused by a body)*} A noninstance body other than a renames-as-body causes freezing of each entity declared before it within the same declarative_part. 3/1

Discussion: This is worded carefully to handle nested packages and private types. Entities declared in a nested package_specification will be frozen by some containing construct. 3.a

An incomplete type declared in the private part of a library package_specification can be completed in the body. 3.b

Ramification: The part about bodies does not say *immediately* within. A renaming-as-body does not have this property. Nor does a pragma Import. 3.c

Reason: The reason bodies cause freezing is because we want proper_bodies and body_stubs to be interchangeable — one should be able to move a proper_body to a subunit, and vice-versa, without changing the semantics. Clearly, anything that should cause freezing should do so even if it's inside a proper_body. However, if we make it a body_stub, then the compiler can't see that thing that should cause freezing. So we make body_stubs cause freezing, just in case they contain something that should cause freezing. But that means we need to do the same for proper_bodies. 3.d

Another reason for bodies to cause freezing, there could be an added implementation burden if an entity declared in an enclosing declarative_part is frozen within a nested body, since some compilers look at bodies after looking at the containing declarative_part. 3.e

{*8652/0046*} {*AI95-00106-01*} {*freezing (entity caused by a construct)* [distributed]} A construct that (explicitly or implicitly) references an entity can cause the *freezing* of the entity, as defined by subsequent paragraphs. {*freezing (by a constituent of a construct)* [partial]} At the place where a construct causes freezing, each name, expression, implicit_dereference[, or range] within the construct causes freezing: 4/1

Ramification: Note that in the sense of this paragraph, a subtype_mark "references" the denoted subtype, but not the type. 4.a

- {*freezing (generic_instantiation)* [partial]} The occurrence of a generic_instantiation causes freezing; also, if a parameter of the instantiation is defaulted, the default_expression or default_name for that parameter causes freezing. 5
- {*freezing (object_declaration)* [partial]} The occurrence of an object_declaration that has no corresponding completion causes freezing. 6

Ramification: Note that this does not include a formal_object_declaration. 6.a

- {*freezing (subtype caused by a record extension)* [partial]} The declaration of a record extension causes freezing of the parent subtype. 7

Ramification: This combined with another rule specifying that primitive subprogram declarations shall precede freezing ensures that all descendants of a tagged type implement all of its dispatching operations. 7.a

{*AI95-00251-01*} The declaration of a private extension does not cause freezing. The freezing is deferred until the full type declaration, which will necessarily be for a record extension, task, or protected type (the latter only for a limited private extension derived from an interface). 7.b/2

7.1/2 • {*AI95-00251-01*} The declaration of a record extension, interface type, task unit, or protected unit causes freezing of any progenitor types specified in the declaration.

7.b.1/2 **Reason:** This rule has the same purpose as the one above: ensuring that all descendants of an interface tagged type implement all of its dispatching operations. As with the previous rule, a private extension does not freeze its progenitors; the full type declaration (which must have the same progenitors) will do that.

7.b.2/2 **Ramification:** An interface type can be a parent as well as a progenitor; these rules are similar so that the location of an interface in a record extension does not have an effect on the freezing of the interface type.

8/1 {*8652/0046*} {*AI95-00106-01*} {*freezing (by an expression)* [partial]} A static expression causes freezing where it occurs. {*freezing (by an object name)* [partial]} An object name or nonstatic expression causes freezing where it occurs, unless the name or expression is part of a default_expression, a default_name, or a per-object expression of a component's constraint, in which case, the freezing occurs later as part of another construct.

8.1/1 {*8652/0046*} {*AI95-00106-01*} {*freezing (by an implicit call)* [partial]} An implicit call freezes the same entities that would be frozen by an explicit call. This is true even if the implicit call is removed via implementation permissions.

8.2/1 {*8652/0046*} {*AI95-00106-01*} {*freezing (subtype caused by an implicit conversion)* [partial]} If an expression is implicitly converted to a type or subtype *T*, then at the place where the expression causes freezing, *T* is frozen.

9 The following rules define which entities are frozen at the place where a construct causes freezing:

10 • {*freezing (type caused by an expression)* [partial]} At the place where an expression causes freezing, the type of the expression is frozen, unless the expression is an enumeration literal used as a discrete_choice of the array_aggregate of an enumeration_representation_clause.

10.a **Reason:** We considered making enumeration literals never cause freezing, which would be more upward compatible, but examples like the variant record aggregate (Discrim => Red, ...) caused us to change our mind. Furthermore, an enumeration literal is a static expression, so the implementation should be allowed to represent it using its representation.

10.b **Ramification:** The following pathological example was legal in Ada 83, but is illegal in Ada 95:

10.c
```
package P1 is
    type T is private;
    package P2 is
        type Composite(D : Boolean) is
            record
                case D is
                    when False => Cf : Integer;
                    when True  => Ct : T;
                end case;
            end record;
    end P2;
    X : Boolean := P2."="( (False,1), (False,1) );
private
    type T is array(1..Func_Call) of Integer;
end;
```

10.d In Ada 95, the declaration of X freezes Composite (because it contains an expression of that type), which in turn freezes T (even though Ct does not exist in this particular case). But type T is not completely defined at that point, violating the rule that a type shall be completely defined before it is frozen. In Ada 83, on the other hand, there is no occurrence of the name T, hence no forcing occurrence of T.

11 • {*freezing (entity caused by a name)* [partial]} At the place where a name causes freezing, the entity denoted by the name is frozen, unless the name is a prefix of an expanded name; {*freezing (nominal subtype caused by a name)* [partial]} at the place where an object name causes freezing, the nominal subtype associated with the name is frozen.

11.a/2 **Ramification:** {*AI95-00114-01*} This only matters in the presence of deferred constants or access types; an object_declaration other than a deferred constant declaration causes freezing of the nominal subtype, plus all component junk.

11.b/1 *This paragraph was deleted.*{*8652/0046*} {*AI95-00106-01*}

- {*8652/0046*} {*AI95-00106-01*} {*freezing (subtype caused by an implicit dereference)* [partial]} At the place where an implicit_dereference causes freezing, the nominal subtype associated with the implicit_dereference is frozen. 11.1/1

  Discussion: This rule ensures that X.D freezes the same entities that X.**all**.D does. Note that an implicit_dereference is neither a name nor expression by itself, so it isn't covered by other rules. 11.c/2

- [{*freezing (type caused by a range)* [partial]} At the place where a range causes freezing, the type of the range is frozen.] 12

  Proof: This is consequence of the facts that expressions freeze their type, and the Range attribute is defined to be equivalent to a pair of expressions separated by "..".} 12.a

- {*freezing (designated subtype caused by an allocator)* [partial]} At the place where an allocator causes freezing, the designated subtype of its type is frozen. If the type of the allocator is a derived type, then all ancestor types are also frozen. 13

  Ramification: Allocators also freeze the named subtype, as a consequence of other rules. 13.a

  The ancestor types are frozen to prevent things like this: 13.b

  13.c
  ```
  type Pool_Ptr is access System.Storage_Pools.Root_Storage_Pool'Class;
  function F return Pool_Ptr;
  ```

  13.d
  ```
  package P is
      type A1 is access Boolean;
      type A2 is new A1;
      type A3 is new A2;
      X : A3 := new Boolean; -- Don't know what pool yet!
      for A1'Storage_Pool use F.all;
  end P;
  ```

  This is necessary because derived access types share their parent's pool. 13.e

- {*freezing (subtypes of the profile of a callable entity)* [partial]} At the place where a callable entity is frozen, each subtype of its profile is frozen. If the callable entity is a member of an entry family, the index subtype of the family is frozen. {*freezing (function call)* [partial]} At the place where a function call causes freezing, if a parameter of the call is defaulted, the default_expression for that parameter causes freezing. 14

  Discussion: We don't worry about freezing for procedure calls or entry calls, since a body freezes everything that precedes it, and the end of a declarative part freezes everything in the declarative part. 14.a

- {*freezing (type caused by the freezing of a subtype)* [partial]} At the place where a subtype is frozen, its type is frozen. {*freezing (constituents of a full type definition)* [partial]} {*freezing (first subtype caused by the freezing of the type)* [partial]} At the place where a type is frozen, any expressions or names within the full type definition cause freezing; the first subtype, and any component subtypes, index subtypes, and parent subtype of the type are frozen as well. {*freezing (class-wide type caused by the freezing of the specific type)* [partial]} {*freezing (specific type caused by the freezing of the class-wide type)* [partial]} For a specific tagged type, the corresponding class-wide type is frozen as well. For a class-wide type, the corresponding specific type is frozen as well. 15

  Ramification: Freezing a type needs to freeze its first subtype in order to preserve the property that the subtype-specific aspects of statically matching subtypes are the same. 15.a

  Freezing an access type does not freeze its designated subtype. 15.b

- {*AI95-00341-01*} At the place where a specific tagged type is frozen, the primitive subprograms of the type are frozen. 15.1/2

  Reason: We have a language design principle that all of the details of a specific tagged type are known at its freezing point. But that is only true if the primitive subprograms are frozen at this point as well. Late changes of Import and address clauses violate the principle. 15.c/2

  Implementation Note: This rule means that no implicit call to Initialize or Adjust can freeze a subprogram (the type and thus subprograms would have been frozen at worst at the same point). 15.d/2

Legality Rules

[The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see 3.9.2).] 16

16.a **Reason:** This rule is needed because (1) we don't want people dispatching to things that haven't been declared yet, and (2) we want to allow tagged type descriptors to be static (allocated statically, and initialized to link-time-known symbols). Suppose T2 inherits primitive P from T1, and then overrides P. Suppose P is called *before* the declaration of the overriding P. What should it dispatch to? If the answer is the new P, we've violated the first principle above. If the answer is the old P, we've violated the second principle. (A call to the new one necessarily raises Program_Error, but that's beside the point.)

16.b Note that a call upon a dispatching operation of type *T* will freeze *T*.

16.c We considered applying this rule to all derived types, for uniformity. However, that would be upward incompatible, so we rejected the idea. As in Ada 83, for an untagged type, the above call upon P will call the old P (which is arguably confusing).

17 [A type shall be completely defined before it is frozen (see 3.11.1 and 7.3).]

18 [The completion of a deferred constant declaration shall occur before the constant is frozen (see 7.4).]

18.a/2 **Proof:** {*AI95-00114-01*} The above Legality Rules are stated “officially” in the referenced clauses.

19/1 {*8652/0009*} {*AI95-00137-01*} An operational or representation item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.1).

19.a/1 **Discussion:** {*8652/0009*} {*AI95-00137-01*} From RM83-13.1(7). The wording here forbids freezing within the aspect_clause itself, which was not true of the Ada 83 wording. The wording of this rule is carefully written to work properly for type-related representation items. For example, an enumeration_representation_clause is illegal after the type is frozen, even though the _clause refers to the first subtype.

19.a.1/2 {*AI95-00114-01*} The above Legality Rule is stated for types and subtypes in 13.1, but the rule here covers all other entities as well.

19.b/2 *This paragraph was deleted.*{*AI95-00114-01*} .

19.c **Discussion:** Here's an example that illustrates when freezing occurs in the presence of defaults:

19.d
```
type T is ...;
function F return T;
type R is
    record
        C : T := F;
        D : Boolean := F = F;
    end record;
X : R;
```

19.e Since the elaboration of R's declaration does not allocate component C, there is no need to freeze C's subtype at that place. Similarly, since the elaboration of R does not evaluate the default_expression “F = F”, there is no need to freeze the types involved at that point. However, the declaration of X *does* need to freeze these things. Note that even if component C did not exist, the elaboration of the declaration of X would still need information about T — even though D is not of type T, its default_expression requires that information.

19.f **Ramification:** Although we define freezing in terms of the program text as a whole (i.e. after applying the rules of Section 10), the freezing rules actually have no effect beyond compilation unit boundaries.

19.g **Reason:** That is important, because Section 10 allows some implementation definedness in the order of things, and we don't want the freezing rules to be implementation defined.

19.h **Ramification:** These rules also have no effect in statements — they only apply within a single declarative_part, package_specification, task_definition, protected_definition, or protected_body.

19.i **Implementation Note:** An implementation may choose to generate code for default_expressions and default_names in line at the place of use. {*thunk*} Alternatively, an implementation may choose to generate thunks (subprograms implicitly generated by the compiler) for evaluation of defaults. Thunk generation cannot, in general, be done at the place of the declaration that includes the default. Instead, they can be generated at the first freezing point of the type(s) involved. (It is impossible to write a purely one-pass Ada compiler, for various reasons. This is one of them — the compiler needs to store a representation of defaults in its symbol table, and then walk that representation later, no earlier than the first freezing point.)

19.j In implementation terms, the linear elaboration model can be thought of as preventing uninitialized dope. For example, the implementation might generate dope to contain the size of a private type. This dope is initialized at the place where the type becomes completely defined. It cannot be initialized earlier, because of the order-of-elaboration rules. The freezing rules prevent elaboration of earlier declarations from accessing the size dope for a private type before it is initialized.

19.k 2.8 overrides the freezing rules in the case of unrecognized pragmas.

{*8652/0009*} {*AI95-00137-01*} An aspect_clause for an entity should most certainly *not* be a freezing point for the entity. 19.l/1

Dynamic Semantics

{*AI95-00279-01*} The tag (see 3.9) of a tagged type T is created at the point where T is frozen.{*creation (of a tag)* [partial]} 20/2

Incompatibilities With Ada 83

{*incompatibilities with Ada 83*} RM83 defines a forcing occurrence of a type as follows: "A forcing occurrence is any occurrence [of the name of the type, subtypes of the type, or types or subtypes with subcomponents of the type] other than in a type or subtype declaration, a subprogram specification, an entry declaration, a deferred constant declaration, a pragma, or a representation_clause for the type itself. In any case, an occurrence within an expression is always forcing." 20.a

It seems like the wording allows things like this: 20.b

20.c
```
type A is array(Integer range 1..10) of Boolean;
subtype S is Integer range A'Range;
    -- not forcing for A
```

Occurrences within pragmas can cause freezing in Ada 95. (Since such pragmas are ignored in Ada 83, this will probably fix more bugs than it causes.) 20.d

Extensions to Ada 83

{*extensions to Ada 83*} In Ada 95, generic_formal_parameter_declarations do not normally freeze the entities from which they are defined. For example: 20.e

20.f
```
package Outer is
    type T is tagged limited private;
    generic
        type T2 is
            new T with private; -- Does not freeze T
                                -- in Ada 95.
    package Inner is
        ...
    end Inner;
private
    type T is ...;
end Outer;
```

This is important for the usability of generics. The above example uses the Ada 95 feature of formal derived types. Examples using the kinds of formal parameters already allowed in Ada 83 are well known. See, for example, comments 83-00627 and 83-00688. The extensive use expected for formal derived types makes this issue even more compelling than described by those comments. Unfortunately, we are unable to solve the problem that explicit_generic_actual_parameters cause freezing, even though a package equivalent to the instance would not cause freezing. This is primarily because such an equivalent package would have its body in the body of the containing program unit, whereas an instance has its body right there. 20.g

Wording Changes from Ada 83

The concept of freezing is based on Ada 83's concept of "forcing occurrences." The first freezing point of an entity corresponds roughly to the place of the first forcing occurrence, in Ada 83 terms. The reason for changing the terminology is that the new rules do not refer to any particular "occurrence" of a name of an entity. Instead, we refer to "uses" of an entity, which are sometimes implicit. 20.h

In Ada 83, forcing occurrences were used only in rules about representation_clauses. We have expanded the concept to cover private types, because the rules stated in RM83-7.4.1(4) are almost identical to the forcing occurrence rules. 20.i

The Ada 83 rules are changed in Ada 95 for the following reasons: 20.j

- The Ada 83 rules do not work right for subtype-specific aspects. In an earlier version of Ada 9X, we considered allowing representation items to apply to subtypes other than the first subtype. This was part of the reason for changing the Ada 83 rules. However, now that we have dropped that functionality, we still need the rules to be different from the Ada 83 rules. 20.k
- The Ada 83 rules do not achieve the intended effect. In Ada 83, either with or without the AIs, it is possible to force the compiler to generate code that references uninitialized dope, or force it to detect erroneousness and exception raising at compile time. 20.l
- It was a goal of Ada 83 to avoid uninitialized access values. However, in the case of deferred constants, this goal was not achieved. 20.m

20.n
- The Ada 83 rules are not only too weak — they are also too strong. They allow loopholes (as described above), but they also prevent certain kinds of default_expressions that are harmless, and certain kinds of generic_declarations that are both harmless and very useful.

20.o/2
- {*AI95-00114-01*} Ada 83 had a case where a aspect_clause had a strong effect on the semantics of the program — 'Small. This caused certain semantic anomalies. There are more cases in Ada 95, because the attribute_definition_clause has been generalized.

Incompatibilities With Ada 95

20.p/2 {*8652/0046*} {*AI95-00106-01*} {*AI95-00341-01*} {*incompatibilities with Ada 95*} **Corrigendum:** Various freezing rules were added to fix holes in the rules. Most importantly, implicit calls are now freezing, which make some representation clauses illegal in Ada 2005 that were legal (but dubious) in Ada 95. **Amendment Correction:** Similarly, the primitive subprograms of a specific tagged type are frozen when the type is frozen, preventing dubious convention changes (and address clauses) after the freezing point. In both cases, the code is dubious and the workaround is easy.

Wording Changes from Ada 95

20.q/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added wording to specify that both operational and representation attributes must be specified before the type is frozen.

20.r/2 {*AI95-00251-01*} Added wording that declaring a specific descendant of an interface type freezes the interface type.

20.s/2 {*AI95-00279-01*} Added wording that defines when a tag is created for a type (at the freezing point of the type). This is used to specify checking for uncreated tags (see 3.9).

The Standard Libraries

Annex A
(normative)
Predefined Language Environment

1 [{*Language-Defined Library Units*} {*predefined environment*} This Annex contains the specifications of library units that shall be provided by every implementation. There are three root library units: Ada, Interfaces, and System; other library units are children of these:]

2/2 {*8652/0047*} {*AI95-00081-01*} {*AI95-00424-01*}

- [Standard — A.1
  - Ada — A.2
    - Assertions — 11.4.2
    - Asynchronous_Task_Control — D.11
    - Calendar — 9.6
      - Arithmetic — 9.6.1
      - Formatting — 9.6.1
      - Time_Zones — 9.6.1
    - Characters — A.3.1
      - Conversions — A.3.4
      - Handling — A.3.2
      - Latin_1 — A.3.3
    - Command_Line — A.15
    - Complex_Text_IO — G.1.3
    - Containers — A.18.1
      - Doubly_Linked_Lists — A.18.3
      - Generic_Array_Sort — A.18.16
      - Generic_Constrained_Array_Sort — A.18.16
      - Hashed_Maps — A.18.5
      - Hashed_Sets — A.18.8
      - Indefinite_Doubly_Linked_Lists — A.18.11
      - Indefinite_Hashed_Maps — A.18.12
      - Indefinite_Hashed_Sets — A.18.14
      - Indefinite_Ordered_Maps — A.18.13
      - Indefinite_Ordered_Sets — A.18.15
      - Indefinite_Vectors — A.18.10
      - Ordered_Maps — A.18.6
      - Ordered_Sets — A.18.9
      - Vectors — A.18.2
    - Decimal — F.2
    - Direct_IO — A.8.4
    - Directories — A.16
      - Information — A.16
    - Dispatching — D.2.1
      - EDF — D.2.6
      - Round_Robin — D.2.5
    - Dynamic_Priorities — D.5

- Standard (...*continued*)
  - Ada (...*continued*)
    - Environment_Variables — A.17
    - Exceptions — 11.4.1
    - Execution_Time — D.14
      - Group_Budgets — D.14.2
      - Timers — D.14.1
    - Finalization — 7.6
    - Float_Text_IO — A.10.9
    - Float_Wide_Text_IO — A.11
    - Float_Wide_Wide_Text_IO — A.11
    - Integer_Text_IO — A.10.8
    - Integer_Wide_Text_IO — A.11
    - Integer_Wide_Wide_Text_IO — A.11
    - Interrupts — C.3.2
      - Names — C.3.2
    - IO_Exceptions — A.13
    - Numerics — A.5
      - Complex_Arrays — G.3.2
      - Complex_Elementary_Functions — G.1.2
      - Complex_Types — G.1.1
      - Discrete_Random — A.5.2
      - Elementary_Functions — A.5.1
      - Float_Random — A.5.2
      - Generic_Complex_Arrays — G.3.2
      - Generic_Complex_Elementary_Functions — G.1.2
      - Generic_Complex_Types — G.1.1
      - Generic_Elementary_Functions — A.5.1
      - Generic_Real_Arrays — G.3.1
      - Real_Arrays — G.3.1
    - Real_Time — D.8
      - Timing_Events — D.15
    - Sequential_IO — A.8.1
    - Storage_IO — A.9
    - Streams — 13.13.1
      - Stream_IO — A.12.1

Standard (...*continued*)
  Ada (...*continued*)
    Strings — A.4.1
      Bounded — A.4.4
        Hash — A.4.9
      Fixed — A.4.3
        Hash — A.4.9
      Hash — A.4.9
      Maps — A.4.2
        Constants — A.4.6
      Unbounded — A.4.5
        Hash — A.4.9
      Wide_Bounded — A.4.7
        Wide_Hash — A.4.7
      Wide_Fixed — A.4.7
        Wide_Hash — A.4.7
      Wide_Hash — A.4.7
      Wide_Maps — A.4.7
        Wide_Constants — A.4.7
      Wide_Unbounded — A.4.7
        Wide_Hash — A.4.7
      Wide_Wide_Bounded — A.4.8
        Wide_Wide_Hash — A.4.8
      Wide_Wide_Fixed — A.4.8
        Wide_Wide_Hash — A.4.8
      Wide_Wide_Hash — A.4.8
      Wide_Wide_Maps — A.4.8
        Wide_Wide_Constants — A.4.8
      Wide_Wide_Unbounded — A.4.8
        Wide_Wide_Hash — A.4.8
    Synchronous_Task_Control — D.10
    Tags — 3.9
      Generic_Dispatching_Constructor — 3.9
    Task_Attributes — C.7.2
    Task_Identification — C.7.1
    Task_Termination — C.7.3

Standard (...*continued*)
  Ada (...*continued*)
    Text_IO — A.10.1
      Bounded_IO — A.10.11
      Complex_IO — G.1.3
      Editing — F.3.3
      Text_Streams — A.12.2
      Unbounded_IO — A.10.12
    Unchecked_Conversion — 13.9
    Unchecked_Deallocation — 13.11.2
    Wide_Characters — A.3.1
    Wide_Text_IO — A.11
      Complex_IO — G.1.4
      Editing — F.3.4
      Text_Streams — A.12.3
      Wide_Bounded_IO — A.11
      Wide_Unbounded_IO — A.11
    Wide_Wide_Characters — A.3.1
    Wide_Wide_Text_IO — A.11
      Complex_IO — G.1.5
      Editing — F.3.5
      Text_Streams — A.12.4
      Wide_Wide_Bounded_IO — A.11
      Wide_Wide_Unbounded_IO — A.11

  Interfaces — B.2
    C — B.3
      Pointers — B.3.2
      Strings — B.3.1
    COBOL — B.4
    Fortran — B.5

  System — 13.7
    Address_To_Access_Conversions — 13.7.2
    Machine_Code — 13.8
    RPC — E.5
    Storage_Elements — 13.7.1
    Storage_Pools — 13.11]

2.a **Discussion:** In running text, we generally leave out the “Ada.” when referring to a child of Ada.

2.b **Reason:** We had no strict rule for which of Ada, Interfaces, or System should be the parent of a given library unit. However, we have tried to place as many things as possible under Ada, except that interfacing is a separate category, and we have tried to place library units whose use is highly non-portable under System.

Implementation Requirements

3/2 {*AI95-00434-01*} The implementation shall ensure that each language-defined subprogram is reentrant{*reentrant*} in the sense that concurrent calls on the same subprogram perform as specified, so long as all parameters that could be passed by reference denote nonoverlapping objects.

3.a **Ramification:** For example, simultaneous calls to Text_IO.Put will work properly, so long as they are going to two different files. On the other hand, simultaneous output to the same file constitutes erroneous use of shared variables.

3.b **To be honest:** Here, “language defined subprogram” means a language defined library subprogram, a subprogram declared in the visible part of a language defined library package, an instance of a language defined generic library subprogram, or a subprogram declared in the visible part of an instance of a language defined generic library package.

3.c **Ramification:** The rule implies that any data local to the private part or body of the package has to be somehow protected against simultaneous access.

Implementation Permissions

The implementation may restrict the replacement of language-defined compilation units. The implementation may restrict children of language-defined library units (other than Standard). 4

Ramification: For example, the implementation may say, “you cannot compile a library unit called System” or “you cannot compile a child of package System” or “if you compile a library unit called System, it has to be a package, and it has to contain at least the following declarations: ...”. 4.a

Wording Changes from Ada 83

Many of Ada 83's language-defined library units are now children of Ada or System. For upward compatibility, these are renamed as root library units (see J.1). 4.b

The order and lettering of the annexes has been changed. 4.c

Wording Changes from Ada 95

{*8652/0047*} {*AI95-00081-01*} **Corrigendum:** Units missing from the list of predefined units were added. 4.d/2

A.1 The Package Standard

This clause outlines the specification of the package Standard containing all predefined identifiers in the language. {*unspecified* [partial]} The corresponding package body is not specified by the language. 1

The operators that are predefined for the types declared in the package Standard are given in comments since they are implicitly declared. {*italics (pseudo-names of anonymous types)*} Italics are used for pseudo-names of anonymous types (such as *root_real*) and for undefined information (such as *implementation-defined*). 2

Ramification: All of the predefined operators are of convention Intrinsic. 2.a

Static Semantics

The library package Standard has the following declaration: 3

Implementation defined: The names and characteristics of the numeric subtypes declared in the visible part of package Standard. 3.a

```
package Standard is                                                          4
   pragma Pure(Standard);

   type Boolean is (False, True);                                            5

   -- The predefined relational operators for this type are as follows:      6

{8652/0028} {AI95-00145-01}    -- function "="   (Left, Right : Boolean'Base)   7/1
return Boolean;
   -- function "/="  (Left, Right : Boolean'Base) return Boolean;
   -- function "<"   (Left, Right : Boolean'Base) return Boolean;
   -- function "<="  (Left, Right : Boolean'Base) return Boolean;
   -- function ">"   (Left, Right : Boolean'Base) return Boolean;
   -- function ">="  (Left, Right : Boolean'Base) return Boolean;

   -- The predefined logical operators and the predefined logical            8
   -- negation operator are as follows:

{8652/0028} {AI95-00145-01}    -- function "and" (Left, Right : Boolean'Base)   9/1
return Boolean'Base;
   -- function "or"  (Left, Right : Boolean'Base) return Boolean'Base;
   -- function "xor" (Left, Right : Boolean'Base) return Boolean'Base;

{8652/0028} {AI95-00145-01}    -- function "not" (Right : Boolean'Base) return   10/1
Boolean'Base;

{AI95-00434-01}       -- The integer type root_integer and the               11/2
   -- corresponding universal type universal_integer are predefined.

   type Integer is range implementation-defined;                             12
```

13
```
   subtype Natural  is Integer range 0 .. Integer'Last;
   subtype Positive is Integer range 1 .. Integer'Last;
```
14
```
   -- The predefined operators for type Integer are as follows:
```
15
```
   -- function "="  (Left, Right : Integer'Base) return Boolean;
   -- function "/=" (Left, Right : Integer'Base) return Boolean;
   -- function "<"  (Left, Right : Integer'Base) return Boolean;
   -- function "<=" (Left, Right : Integer'Base) return Boolean;
   -- function ">"  (Left, Right : Integer'Base) return Boolean;
   -- function ">=" (Left, Right : Integer'Base) return Boolean;
```
16
```
   -- function "+"   (Right : Integer'Base) return Integer'Base;
   -- function "-"   (Right : Integer'Base) return Integer'Base;
   -- function "abs" (Right : Integer'Base) return Integer'Base;
```
17
```
   -- function "+"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "-"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "*"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "/"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "rem" (Left, Right : Integer'Base) return Integer'Base;
   -- function "mod" (Left, Right : Integer'Base) return Integer'Base;
```
18
```
   -- function "**"  (Left : Integer'Base; Right : Natural)
   --                  return Integer'Base;
```
19
```
   -- The specification of each operator for the type
   -- root_integer, or for any additional predefined integer
   -- type, is obtained by replacing Integer by the name of the type
   -- in the specification of the corresponding operator of the type
   -- Integer. The right operand of the exponentiation operator
   -- remains as subtype Natural.
```
20/2
```
{AI95-00434-01}       -- The floating point type root_real and the
   -- corresponding universal type universal_real are predefined.
```
21
```
   type Float is digits implementation-defined;
```
22
```
   -- The predefined operators for this type are as follows:
```
23
```
   -- function "="   (Left, Right : Float) return Boolean;
   -- function "/="  (Left, Right : Float) return Boolean;
   -- function "<"   (Left, Right : Float) return Boolean;
   -- function "<="  (Left, Right : Float) return Boolean;
   -- function ">"   (Left, Right : Float) return Boolean;
   -- function ">="  (Left, Right : Float) return Boolean;
```
24
```
   -- function "+"   (Right : Float) return Float;
   -- function "-"   (Right : Float) return Float;
   -- function "abs" (Right : Float) return Float;
```
25
```
   -- function "+"   (Left, Right : Float) return Float;
   -- function "-"   (Left, Right : Float) return Float;
   -- function "*"   (Left, Right : Float) return Float;
   -- function "/"   (Left, Right : Float) return Float;
```
26
```
   -- function "**"  (Left : Float; Right : Integer'Base) return Float;
```
27
```
   -- The specification of each operator for the type root_real, or for
   -- any additional predefined floating point type, is obtained by
   -- replacing Float by the name of the type in the specification of the
   -- corresponding operator of the type Float.
```
28
```
   -- In addition, the following operators are predefined for the root
   -- numeric types:
```
29
```
   function "*" (Left : root_integer; Right : root_real)
     return root_real;
```
30
```
   function "*" (Left : root_real;     Right : root_integer)
     return root_real;
```
31
```
   function "/" (Left : root_real;     Right : root_integer)
     return root_real;
   -- The type universal_fixed is predefined.                                   32
   -- The only multiplying operators defined between
   -- fixed point types are

   function "*" (Left : universal_fixed; Right : universal_fixed)               33
     return universal_fixed;

   function "/" (Left : universal_fixed; Right : universal_fixed)               34
     return universal_fixed;

{AI95-00230-01}     -- The type universal_access is predefined.                 34.1/2
   -- The following equality operators are predefined:

{AI95-00230-01}     function "="  (Left, Right: universal_access) return Boolean;   34.2/2
   function "/=" (Left, Right: universal_access) return Boolean;
```

35/2 *{AI95-00415-01}* *-- The declaration of type Character is based on the standard ISO 8859-1 character set.*

```
    -- There are no character literals corresponding to the positions for control characters.
    -- They are indicated in italics in this definition. See 3.5.2.

type Character is
  (nul,   soh,   stx,   etx,      eot,   enq,   ack,   bel,   --0 (16#00#) .. 7 (16#07#)
   bs,    ht,    lf,    vt,       ff,    cr,    so,    si,    --8 (16#08#) .. 15 (16#0F#)

   dle,   dc1,   dc2,   dc3,      dc4,   nak,   syn,   etb,   --16 (16#10#) .. 23 (16#17#)
   can,   em,    sub,   esc,      fs,    gs,    rs,    us,    --24 (16#18#) .. 31 (16#1F#)

   ' ',   '!',   '"',   '#',      '$',   '%',   '&',   ''',   --32 (16#20#) .. 39 (16#27#)
   '(',   ')',   '*',   '+',      ',',   '-',   '.',   '/',   --40 (16#28#) .. 47 (16#2F#)

   '0',   '1',   '2',   '3',      '4',   '5',   '6',   '7',   --48 (16#30#) .. 55 (16#37#)
   '8',   '9',   ':',   ';',      '<',   '=',   '>',   '?',   --56 (16#38#) .. 63 (16#3F#)

   '@',   'A',   'B',   'C',      'D',   'E',   'F',   'G',   --64 (16#40#) .. 71 (16#47#)
   'H',   'I',   'J',   'K',      'L',   'M',   'N',   'O',   --72 (16#48#) .. 79 (16#4F#)

   'P',   'Q',   'R',   'S',      'T',   'U',   'V',   'W',   --80 (16#50#) .. 87 (16#57#)
   'X',   'Y',   'Z',   '[',      '\',   ']',   '^',   '_',   --88 (16#58#) .. 95 (16#5F#)

   '`',   'a',   'b',   'c',      'd',   'e',   'f',   'g',   --96 (16#60#) .. 103 (16#67#)
   'h',   'i',   'j',   'k',      'l',   'm',   'n',   'o',   --104 (16#68#) .. 111 (16#6F#)

   'p',   'q',   'r',   's',      't',   'u',   'v',   'w',   --112 (16#70#) .. 119 (16#77#)
   'x',   'y',   'z',   '{',      '|',   '}',   '~',   del,   --120 (16#78#) .. 127 (16#7F#)

   reserved_128, reserved_129,    bph,   nbh,                 --128 (16#80#) .. 131 (16#83#)
   reserved_132, nel,   ssa,      esa,                        --132 (16#84#) .. 135 (16#87#)
   hts,   htj,   vts,   pld,      plu,   ri,    ss2,   ss3,   --136 (16#88#) .. 143 (16#8F#)

   dcs,   pu1,   pu2,   sts,      cch,   mw,    spa,   epa,   --144 (16#90#) .. 151 (16#97#)
   sos,   reserved_153, sci,      csi,                        --152 (16#98#) .. 155 (16#9B#)
   st,    osc,   pm,    apc,                                  --156 (16#9C#) .. 159 (16#9F#)

   ' ',   '¡',   '¢',   '£',      '¤',   '¥',   '¦',   '§',   --160 (16#A0#) .. 167 (16#A7#)
   '¨',   '©',   'ª',   '«',      '¬',   '-',   '®',   '¯',   --168 (16#A8#) .. 175 (16#AF#)

   '°',   '±',   '²',   '³',      '´',   'µ',   '¶',   '·',   --176 (16#B0#) .. 183 (16#B7#)
   '¸',   '¹',   'º',   '»',      '¼',   '½',   '¾',   '¿',   --184 (16#B8#) .. 191 (16#BF#)

   'À',   'Á',   'Â',   'Ã',      'Ä',   'Å',   'Æ',   'Ç',   --192 (16#C0#) .. 199 (16#C7#)
   'È',   'É',   'Ê',   'Ë',      'Ì',   'Í',   'Î',   'Ï',   --200 (16#C8#) .. 207 (16#CF#)

   'Ð',   'Ñ',   'Ò',   'Ó',      'Ô',   'Õ',   'Ö',   '×',   --208 (16#D0#) .. 215 (16#D7#)
   'Ø',   'Ù',   'Ú',   'Û',      'Ü',   'Ý',   'Þ',   'ß',   --216 (16#D8#) .. 223 (16#DF#)

   'à',   'á',   'â',   'ã',      'ä',   'å',   'æ',   'ç',   --224 (16#E0#) .. 231 (16#E7#)
   'è',   'é',   'ê',   'ë',      'ì',   'í',   'î',   'ï',   --232 (16#E8#) .. 239 (16#EF#)

   'ð',   'ñ',   'ò',   'ó',      'ô',   'õ',   'ö',   '÷',   --240 (16#F0#) .. 247 (16#F7#)
   'ø',   'ù',   'ú',   'û',      'ü',   'ý',   'þ',   'ÿ');--248 (16#F8#) .. 255 (16#FF#)
```

36
```
-- The predefined operators for the type Character are the same as for
-- any enumeration type.
```

36.1/2 *{AI95-00395-01}* *-- The declaration of type Wide_Character is based on the standard ISO/IEC 10646:2003 BMP character*

```
    -- set. The first 256 positions have the same contents as type Character. See 3.5.2.

type Wide_Character is (nul, soh ... Hex_0000FFFE, Hex_0000FFFF);
```

```
{AI95-00285-01} {AI95-00395-01}       -- The declaration of type Wide_Wide_Character is based on the full     36.2/2
    -- ISO/IEC 10646:2003 character set. The first 65536 positions have the
    -- same contents as type Wide_Character. See 3.5.2.

    type Wide_Wide_Character is (nul, soh ... Hex_7FFFFFFE, Hex_7FFFFFFF);
    for Wide_Wide_Character'Size use 32;

    package ASCII is ... end ASCII;  --Obsolescent; see J.5                                                  36.3/2
{ASCII (package physically nested within the declaration of Standard)}

    -- Predefined string types:                                                                              37

    type String is array(Positive range <>) of Character;
    pragma Pack(String);

    -- The predefined operators for this type are as follows:                                                38

--     function "="  (Left, Right: String) return Boolean;                                                   39
--     function "/=" (Left, Right: String) return Boolean;
--     function "<"  (Left, Right: String) return Boolean;
--     function "<=" (Left, Right: String) return Boolean;
--     function ">"  (Left, Right: String) return Boolean;
--     function ">=" (Left, Right: String) return Boolean;

--     function "&" (Left: String;    Right: String)    return String;                                       40
--     function "&" (Left: Character; Right: String)    return String;
--     function "&" (Left: String;    Right: Character) return String;
--     function "&" (Left: Character; Right: Character) return String;

    type Wide_String is array(Positive range <>) of Wide_Character;                                          41
    pragma Pack(Wide_String);

    -- The predefined operators for this type correspond to those for String.                                42

{AI95-00285-01}    type Wide_Wide_String is array (Positive range <>)                                        42.1/2
      of Wide_Wide_Character;
    pragma Pack (Wide_Wide_String);

{AI95-00285-01}      -- The predefined operators for this type correspond to those for String.               42.2/2

    type Duration is delta implementation-defined range implementation-defined;                              43

       -- The predefined operators for the type Duration are the same as for                                 44
       -- any fixed point type.

    -- The predefined exceptions:                                                                            45

    Constraint_Error: exception;                                                                             46
    Program_Error   : exception;
    Storage_Error   : exception;
    Tasking_Error   : exception;

end Standard;                                                                                                47
```

Standard has no private part. 48

Reason: This is important for portability. All library packages are children of Standard, and if Standard had a private part then it would be visible to all of them. 48.a

{*AI95-00285-01*} In each of the types Character, Wide_Character, and Wide_Wide_Character, the character literals for the space character (position 32) and the non-breaking space character (position 160) correspond to different values. Unless indicated otherwise, each occurrence of the character literal ' ' in this International Standard refers to the space character. Similarly, the character literals for hyphen (position 45) and soft hyphen (position 173) correspond to different values. Unless indicated otherwise, each occurrence of the character literal '–' in this International Standard refers to the hyphen character. 49/2

Dynamic Semantics

{*elaboration (package_body of Standard)* [partial]} Elaboration of the body of Standard has no effect. 50

Discussion: Note that the language does not define where this body appears in the environment declarative_part — see 10, “Program Structure and Compilation Issues”. 50.a

Implementation Permissions

51 An implementation may provide additional predefined integer types and additional predefined floating point types. Not all of these types need have names.

51.a **To be honest:** An implementation may add representation items to package Standard, for example to specify the internal codes of type Boolean, or the Small of type Duration.

Implementation Advice

52 If an implementation provides additional named predefined integer types, then the names should end with "Integer" as in "Long_Integer". If an implementation provides additional named predefined floating point types, then the names should end with "Float" as in "Long_Float".

52.a/2 **Implementation Advice:** If an implementation provides additional named predefined integer types, then the names should end with "Integer". If an implementation provides additional named predefined floating point types, then the names should end with "Float".

NOTES

53 1 Certain aspects of the predefined entities cannot be completely described in the language itself. For example, although the enumeration type Boolean can be written showing the two enumeration literals False and True, the short-circuit control forms cannot be expressed in the language.

54 2 As explained in 8.1, "Declarative Region" and 10.1.4, "The Compilation Process", the declarative region of the package Standard encloses every library unit and consequently the main subprogram; the declaration of every library unit is assumed to occur within this declarative region. Library_items are assumed to be ordered in such a way that there are no forward semantic dependences. However, as explained in 8.3, "Visibility", the only library units that are visible within a given compilation unit are the library units named by all with_clauses that apply to the given unit, and moreover, within the declarative region of a given library unit, that library unit itself.

55 3 If all block_statements of a program are named, then the name of each program unit can always be written as an expanded name starting with Standard (unless Standard is itself hidden). The name of a library unit cannot be a homograph of a name (such as Integer) that is already declared in Standard.

56 4 The exception Standard.Numeric_Error is defined in J.6.

56.a **Discussion:** The declaration of Natural needs to appear between the declaration of Integer and the (implicit) declaration of the "**" operator for Integer, because a formal parameter of "**" is of subtype Natural. This would be impossible in normal code, because the implicit declarations for a type occur immediately after the type declaration, with no possibility of intervening explicit declarations. But we're in Standard, and Standard is somewhat magic anyway.

56.b Using Natural as the subtype of the formal of "**" seems natural; it would be silly to have a textual rule about Constraint_Error being raised when there is a perfectly good subtype that means just that. Furthermore, by not using Integer for that formal, it helps remind the reader that the exponent remains Natural even when the left operand is replaced with the derivative of Integer. It doesn't logically imply that, but it's still useful as a reminder.

56.c In any case, declaring these general-purpose subtypes of Integer close to Integer seems more readable than declaring them much later.

Extensions to Ada 83

56.d {*extensions to Ada 83*} Package Standard is declared to be pure.

56.e **Discussion:** The introduction of the types Wide_Character and Wide_String is not an Ada 95 extension to Ada 83, since ISO WG9 has approved these as an authorized extension of the original Ada 83 standard that is part of that standard.

Wording Changes from Ada 83

56.f Numeric_Error is made obsolescent.

56.g The declarations of Natural and Positive are moved to just after the declaration of Integer, so that "**" can refer to Natural without a forward reference. There's no real need to move Positive, too — it just came along for the ride.

Extensions to Ada 95

56.h/2 {*AI95-00285-01*} {*extensions to Ada 95*} Types Wide_Wide_Character and Wide_Wide_String are new.

56.i/2 **Discussion:** The inconsistencies associated with these types are documented in 3.5.2 and 3.6.3.

56.j/2 {*AI95-00230-01*} Type *universal_access* and the equality operations for it are new.

Wording Changes from Ada 95

{*8652/0028*} {*AI95-00145-01*} **Corrigendum:** Corrected the parameter type for the Boolean operators declared in Standard.. 56.k/2

A.2 The Package Ada

Static Semantics

1 The following language-defined library package exists:

2
```
package Ada is
    pragma Pure(Ada);
end Ada;
```

3 Ada serves as the parent of most of the other language-defined library units; its declaration is empty (except for the pragma Pure).

Legality Rules

4 In the standard mode, it is illegal to compile a child of package Ada.

4.a **Reason:** The intention is that mentioning, say, Ada.Text_IO in a with_clause is guaranteed (at least in the standard mode) to refer to the standard version of Ada.Text_IO. The user can compile a root library unit Text_IO that has no relation to the standard version of Text_IO.

4.b **Ramification:** Note that Ada can have non-language-defined grandchildren, assuming the implementation allows it. Also, packages System and Interfaces can have children, assuming the implementation allows it.

4.c **Implementation Note:** An implementation will typically support a nonstandard mode in which compiling the language defined library units is allowed. Whether or not this mode is made available to users is up to the implementer.

4.d An implementation could theoretically have private children of Ada, since that would be semantically neutral. However, a programmer cannot compile such a library unit.

Extensions to Ada 83

4.e {*extensions to Ada 83*} This clause is new to Ada 95.

A.3 Character Handling

1/2 {*AI95-00285-01*} This clause presents the packages related to character processing: an empty pure package Characters and child packages Characters.Handling and Characters.Latin_1. The package Characters.Handling provides classification and conversion functions for Character data, and some simple functions for dealing with Wide_Character and Wide_Wide_Character data. The child package Characters.Latin_1 declares a set of constants initialized to values of type Character.

Extensions to Ada 83

1.a {*extensions to Ada 83*} This clause is new to Ada 95.

Wording Changes from Ada 95

1.b/2 {*AI95-00285-01*} Included Wide_Wide_Character in this description; the individual changes are documented as extensions as needed.

A.3.1 The Packages Characters, Wide_Characters, and Wide_Wide_Characters

Static Semantics

1 The library package Characters has the following declaration:

2
```
package Ada.Characters is
  pragma Pure(Characters);
end Ada.Characters;
```

3/2 {*AI95-00395-01*} The library package Wide_Characters has the following declaration:

4/2
```
package Ada.Wide_Characters is
  pragma Pure(Wide_Characters);
end Ada.Wide_Characters;
```

5/2 {*AI95-00395-01*} The library package Wide_Wide_Characters has the following declaration:

6/2
```
package Ada.Wide_Wide_Characters is
  pragma Pure(Wide_Wide_Characters);
end Ada.Wide_Wide_Characters;
```

Implementation Advice

7/2 {*AI95-00395-01*} If an implementation chooses to provide implementation-defined operations on Wide_Character or Wide_String (such as case mapping, classification, collating and sorting, etc.) it should do so by providing child units of Wide_Characters. Similarly if it chooses to provide implementation-defined operations on Wide_Wide_Character or Wide_Wide_String it should do so by providing child units of Wide_Wide_Characters.

7.a/2 **Implementation Advice:** Implementation-defined operations on Wide_Character, Wide_String, Wide_Wide_Character, and Wide_Wide_String should be child units of Wide_Characters or Wide_Wide_Characters.

Extensions to Ada 95

7.b/2 {*AI95-00395-01*} {*extensions to Ada 95*} The packages Wide_Characters and Wide_Wide_Characters are new.

A.3.2 The Package Characters.Handling

Static Semantics

1 The library package Characters.Handling has the following declaration:

2/2
```
{AI95-00362-01} {AI95-00395-01} with Ada.Characters.Conversions;
package Ada.Characters.Handling is
  pragma Pure(Handling);
```

3
```
--Character classification functions
```

4
```
  function Is_Control           (Item : in Character) return Boolean;
  function Is_Graphic           (Item : in Character) return Boolean;
  function Is_Letter            (Item : in Character) return Boolean;
  function Is_Lower             (Item : in Character) return Boolean;
  function Is_Upper             (Item : in Character) return Boolean;
  function Is_Basic             (Item : in Character) return Boolean;
  function Is_Digit             (Item : in Character) return Boolean;
  function Is_Decimal_Digit     (Item : in Character) return Boolean
                     renames Is_Digit;
  function Is_Hexadecimal_Digit (Item : in Character) return Boolean;
  function Is_Alphanumeric      (Item : in Character) return Boolean;
  function Is_Special           (Item : in Character) return Boolean;
```

```
--Conversion functions for Character and String                                      5

  function To_Lower (Item : in Character) return Character;                           6
  function To_Upper (Item : in Character) return Character;
  function To_Basic (Item : in Character) return Character;

  function To_Lower (Item : in String) return String;                                 7
  function To_Upper (Item : in String) return String;
  function To_Basic (Item : in String) return String;

--Classifications of and conversions between Character and ISO 646                   8

  subtype ISO_646 is                                                                  9
    Character range Character'Val(0) .. Character'Val(127);

  function Is_ISO_646 (Item : in Character) return Boolean;                          10
  function Is_ISO_646 (Item : in String)    return Boolean;

  function To_ISO_646 (Item       : in Character;                                    11
                       Substitute : in ISO_646 := ' ')
    return ISO_646;

  function To_ISO_646 (Item       : in String;                                       12
                       Substitute : in ISO_646 := ' ')
    return String;

{AI95-00285-01} {AI95-00395-01} -- The functions Is_Character, Is_String, To_Character, To_String, To_Wide_Character,   13/2
-- and To_Wide_String are obsolescent; see J.14.

Paragraphs 14 through 18 were deleted.

end Ada.Characters.Handling;                                                         19
```

19.a/2 **Discussion:** {*AI95-00395-01*} The with_clause for Ada.Characters.Conversions is needed for the definition of the obsolescent functions (see J.14). It would be odd to put this clause into J.14 as it was not present in Ada 95, and with_clauses are semantically neutral to clients anyway.

20 In the description below for each function that returns a Boolean result, the effect is described in terms of the conditions under which the value True is returned. If these conditions are not met, then the function returns False.

21 Each of the following classification functions has a formal Character parameter, Item, and returns a Boolean result.

22 {*control character (a category of Character)*} Is_Control
True if Item is a control character. A *control character* is a character whose position is in one of the ranges 0..31 or 127..159.

23 {*graphic character (a category of Character)*} Is_Graphic
True if Item is a graphic character. A *graphic character* is a character whose position is in one of the ranges 32..126 or 160..255.

24 {*letter (a category of Character)*} Is_Letter True if Item is a letter. A *letter* is a character that is in one of the ranges 'A'..'Z' or 'a'..'z', or whose position is in one of the ranges 192..214, 216..246, or 248..255.

25 {*lower-case letter (a category of Character)*} Is_Lower True if Item is a lower-case letter. A *lower-case letter* is a character that is in the range 'a'..'z', or whose position is in one of the ranges 223..246 or 248..255.

26 {*upper-case letter (a category of Character)*} Is_Upper True if Item is an upper-case letter. An *upper-case letter* is a character that is in the range 'A'..'Z' or whose position is in one of the ranges 192..214 or 216.. 222.

27 {*basic letter (a category of Character)*} Is_Basic True if Item is a basic letter. A *basic letter* is a character that is in one of the ranges 'A'..'Z' and 'a'..'z', or that is one of the following: 'Æ', 'æ', 'Ð', 'ð', 'Þ', 'þ', or 'ß'.

28 {*decimal digit (a category of Character)*} Is_Digit True if Item is a decimal digit. A *decimal digit* is a character in the range '0'..'9'.

29 Is_Decimal_Digit
A renaming of Is_Digit.

30 {*hexadecimal digit (a category of Character)*} Is_Hexadecimal_Digit
True if Item is a hexadecimal digit. A *hexadecimal digit* is a character that is either a decimal digit or that is in one of the ranges 'A' .. 'F' or 'a' .. 'f'.

31 {*alphanumeric character (a category of Character)*} Is_Alphanumeric
True if Item is an alphanumeric character. An *alphanumeric character* is a character that is either a letter or a decimal digit.

32 {*special graphic character (a category of Character)*} Is_Special
True if Item is a special graphic character. A *special graphic character* is a graphic character that is not alphanumeric.

33 Each of the names To_Lower, To_Upper, and To_Basic refers to two functions: one that converts from Character to Character, and the other that converts from String to String. The result of each Character-to-Character function is described below, in terms of the conversion applied to Item, its formal Character parameter. The result of each String-to-String conversion is obtained by applying to each element of the function's String parameter the corresponding Character-to-Character conversion; the result is the null String if the value of the formal parameter is the null String. The lower bound of the result String is 1.

34 To_Lower Returns the corresponding lower-case value for Item if Is_Upper(Item), and returns Item otherwise.

35 To_Upper Returns the corresponding upper-case value for Item if Is_Lower(Item) and Item has an upper-case form, and returns Item otherwise. The lower case letters 'ß' and 'ÿ' do not have upper case forms.

36 To_Basic Returns the letter corresponding to Item but with no diacritical mark, if Item is a letter but not a basic letter; returns Item otherwise.

37 The following set of functions test for membership in the ISO 646 character range, or convert between ISO 646 and Character.

38 Is_ISO_646
The function whose formal parameter, Item, is of type Character returns True if Item is in the subtype ISO_646.

39 Is_ISO_646
The function whose formal parameter, Item, is of type String returns True if Is_ISO_646(Item(I)) is True for each I in Item'Range.

40 To_ISO_646
The function whose first formal parameter, Item, is of type Character returns Item if Is_ISO_646(Item), and returns the Substitute ISO_646 character otherwise.

41 To_ISO_646
The function whose first formal parameter, Item, is of type String returns the String whose Range is 1..Item'Length and each of whose elements is given by To_ISO_646 of the corresponding element in Item.

Paragraphs 42 through 48 were deleted.

Implementation Advice

49/2 *This paragraph was deleted.*{*AI95-00285-01*}

NOTES
5 A basic letter is a letter without a diacritical mark. 50

6 Except for the hexadecimal digits, basic letters, and ISO_646 characters, the categories identified in the classification functions form a strict hierarchy: 51

— Control characters 52

— Graphic characters 53

— Alphanumeric characters 54

— Letters 55

— Upper-case letters 56

— Lower-case letters 57

— Decimal digits 58

— Special graphic characters 59

Ramification: Thus each Character value is either a control character or a graphic character but not both; each graphic character is either an alphanumeric or special graphic but not both; each alphanumeric is either a letter or decimal digit but not both; each letter is either upper case or lower case but not both. 59.a

Extensions to Ada 95

{*AI95-00362-01*} {*extensions to Ada 95*} Characters.Handling is now Pure, so it can be used in pure units. 59.b/2

Wording Changes from Ada 95

{*AI95-00285-01*} {*AI95-00395-01*} The conversion functions are made obsolescent; a more complete set is available in Characters.Conversions — see A.3.4. 59.c/2

{*AI95-00285-01*} We no longer talk about localized character sets; these are a non-standard mode, which is none of our business. 59.d/2

A.3.3 The Package Characters.Latin_1

The package Characters.Latin_1 declares constants for characters in ISO 8859-1. 1

Reason: The constants for the ISO 646 characters could have been declared as renamings of objects declared in package ASCII, as opposed to explicit constants. The main reason for explicit constants was for consistency of style with the upper-half constants, and to avoid emphasizing the package ASCII. 1.a

Static Semantics

The library package Characters.Latin_1 has the following declaration: 2

```
package Ada.Characters.Latin_1 is                                         3
    pragma Pure(Latin_1);
```

```
-- Control characters:{control character (a category of Character) [partial]}     4
```

```
    NUL                 : constant Character := Character'Val(0);          5
    SOH                 : constant Character := Character'Val(1);
    STX                 : constant Character := Character'Val(2);
    ETX                 : constant Character := Character'Val(3);
    EOT                 : constant Character := Character'Val(4);
    ENQ                 : constant Character := Character'Val(5);
    ACK                 : constant Character := Character'Val(6);
    BEL                 : constant Character := Character'Val(7);
    BS                  : constant Character := Character'Val(8);
    HT                  : constant Character := Character'Val(9);
    LF                  : constant Character := Character'Val(10);
    VT                  : constant Character := Character'Val(11);
    FF                  : constant Character := Character'Val(12);
    CR                  : constant Character := Character'Val(13);
    SO                  : constant Character := Character'Val(14);
    SI                  : constant Character := Character'Val(15);
```

6
```
       DLE                  : constant Character := Character'Val(16);
       DC1                  : constant Character := Character'Val(17);
       DC2                  : constant Character := Character'Val(18);
       DC3                  : constant Character := Character'Val(19);
       DC4                  : constant Character := Character'Val(20);
       NAK                  : constant Character := Character'Val(21);
       SYN                  : constant Character := Character'Val(22);
       ETB                  : constant Character := Character'Val(23);
       CAN                  : constant Character := Character'Val(24);
       EM                   : constant Character := Character'Val(25);
       SUB                  : constant Character := Character'Val(26);
       ESC                  : constant Character := Character'Val(27);
       FS                   : constant Character := Character'Val(28);
       GS                   : constant Character := Character'Val(29);
       RS                   : constant Character := Character'Val(30);
       US                   : constant Character := Character'Val(31);
```

7
```
   -- ISO 646 graphic characters:
```

8
```
       Space                : constant Character := ' ';  -- Character'Val(32)
       Exclamation          : constant Character := '!';  -- Character'Val(33)
       Quotation            : constant Character := '"';  -- Character'Val(34)
       Number_Sign          : constant Character := '#';  -- Character'Val(35)
       Dollar_Sign          : constant Character := '$';  -- Character'Val(36)
       Percent_Sign         : constant Character := '%';  -- Character'Val(37)
       Ampersand            : constant Character := '&';  -- Character'Val(38)
       Apostrophe           : constant Character := ''';  -- Character'Val(39)
       Left_Parenthesis     : constant Character := '(';  -- Character'Val(40)
       Right_Parenthesis    : constant Character := ')';  -- Character'Val(41)
       Asterisk             : constant Character := '*';  -- Character'Val(42)
       Plus_Sign            : constant Character := '+';  -- Character'Val(43)
       Comma                : constant Character := ',';  -- Character'Val(44)
       Hyphen               : constant Character := '-';  -- Character'Val(45)
       Minus_Sign           : Character renames Hyphen;
       Full_Stop            : constant Character := '.';  -- Character'Val(46)
       Solidus              : constant Character := '/';  -- Character'Val(47)
```

9
```
       -- Decimal digits '0' though '9' are at positions 48 through 57
```

10
```
       Colon                : constant Character := ':';  -- Character'Val(58)
       Semicolon            : constant Character := ';';  -- Character'Val(59)
       Less_Than_Sign       : constant Character := '<';  -- Character'Val(60)
       Equals_Sign          : constant Character := '=';  -- Character'Val(61)
       Greater_Than_Sign    : constant Character := '>';  -- Character'Val(62)
       Question             : constant Character := '?';  -- Character'Val(63)
       Commercial_At        : constant Character := '@';  -- Character'Val(64)
```

11
```
       -- Letters 'A' through 'Z' are at positions 65 through 90
```

12
```
       Left_Square_Bracket  : constant Character := '[';  -- Character'Val(91)
       Reverse_Solidus      : constant Character := '\';  -- Character'Val(92)
       Right_Square_Bracket : constant Character := ']';  -- Character'Val(93)
       Circumflex           : constant Character := '^';  -- Character'Val(94)
       Low_Line             : constant Character := '_';  -- Character'Val(95)
```

13
```
       Grave                : constant Character := '`';  -- Character'Val(96)
       LC_A                 : constant Character := 'a';  -- Character'Val(97)
       LC_B                 : constant Character := 'b';  -- Character'Val(98)
       LC_C                 : constant Character := 'c';  -- Character'Val(99)
       LC_D                 : constant Character := 'd';  -- Character'Val(100)
       LC_E                 : constant Character := 'e';  -- Character'Val(101)
       LC_F                 : constant Character := 'f';  -- Character'Val(102)
       LC_G                 : constant Character := 'g';  -- Character'Val(103)
       LC_H                 : constant Character := 'h';  -- Character'Val(104)
       LC_I                 : constant Character := 'i';  -- Character'Val(105)
       LC_J                 : constant Character := 'j';  -- Character'Val(106)
       LC_K                 : constant Character := 'k';  -- Character'Val(107)
       LC_L                 : constant Character := 'l';  -- Character'Val(108)
       LC_M                 : constant Character := 'm';  -- Character'Val(109)
       LC_N                 : constant Character := 'n';  -- Character'Val(110)
       LC_O                 : constant Character := 'o';  -- Character'Val(111)
```

14
```
    LC_P                 : constant Character := 'p';  -- Character'Val(112)
    LC_Q                 : constant Character := 'q';  -- Character'Val(113)
    LC_R                 : constant Character := 'r';  -- Character'Val(114)
    LC_S                 : constant Character := 's';  -- Character'Val(115)
    LC_T                 : constant Character := 't';  -- Character'Val(116)
    LC_U                 : constant Character := 'u';  -- Character'Val(117)
    LC_V                 : constant Character := 'v';  -- Character'Val(118)
    LC_W                 : constant Character := 'w';  -- Character'Val(119)
    LC_X                 : constant Character := 'x';  -- Character'Val(120)
    LC_Y                 : constant Character := 'y';  -- Character'Val(121)
    LC_Z                 : constant Character := 'z';  -- Character'Val(122)
    Left_Curly_Bracket   : constant Character := '{';  -- Character'Val(123)
    Vertical_Line        : constant Character := '|';  -- Character'Val(124)
    Right_Curly_Bracket  : constant Character := '}';  -- Character'Val(125)
    Tilde                : constant Character := '~';  -- Character'Val(126)
    DEL                  : constant Character := Character'Val(127);
```

15
```
-- ISO 6429 control characters:{control character (a category of Character) [partial]}
```

16
```
    IS4                  : Character renames FS;
    IS3                  : Character renames GS;
    IS2                  : Character renames RS;
    IS1                  : Character renames US;
```

17
```
    Reserved_128         : constant Character := Character'Val(128);
    Reserved_129         : constant Character := Character'Val(129);
    BPH                  : constant Character := Character'Val(130);
    NBH                  : constant Character := Character'Val(131);
    Reserved_132         : constant Character := Character'Val(132);
    NEL                  : constant Character := Character'Val(133);
    SSA                  : constant Character := Character'Val(134);
    ESA                  : constant Character := Character'Val(135);
    HTS                  : constant Character := Character'Val(136);
    HTJ                  : constant Character := Character'Val(137);
    VTS                  : constant Character := Character'Val(138);
    PLD                  : constant Character := Character'Val(139);
    PLU                  : constant Character := Character'Val(140);
    RI                   : constant Character := Character'Val(141);
    SS2                  : constant Character := Character'Val(142);
    SS3                  : constant Character := Character'Val(143);
```

18
```
    DCS                  : constant Character := Character'Val(144);
    PU1                  : constant Character := Character'Val(145);
    PU2                  : constant Character := Character'Val(146);
    STS                  : constant Character := Character'Val(147);
    CCH                  : constant Character := Character'Val(148);
    MW                   : constant Character := Character'Val(149);
    SPA                  : constant Character := Character'Val(150);
    EPA                  : constant Character := Character'Val(151);
```

19
```
    SOS                  : constant Character := Character'Val(152);
    Reserved_153         : constant Character := Character'Val(153);
    SCI                  : constant Character := Character'Val(154);
    CSI                  : constant Character := Character'Val(155);
    ST                   : constant Character := Character'Val(156);
    OSC                  : constant Character := Character'Val(157);
    PM                   : constant Character := Character'Val(158);
    APC                  : constant Character := Character'Val(159);
```

20
```
-- Other graphic characters:
```

21
```
-- Character positions 160 (16#A0#) .. 175 (16#AF#):
    No_Break_Space             : constant Character := ' '; --Character'Val(160)
    NBSP                       : Character renames No_Break_Space;
    Inverted_Exclamation       : constant Character := '¡'; --Character'Val(161)
    Cent_Sign                  : constant Character := '¢'; --Character'Val(162)
    Pound_Sign                 : constant Character := '£'; --Character'Val(163)
    Currency_Sign              : constant Character := '¤'; --Character'Val(164)
    Yen_Sign                   : constant Character := '¥'; --Character'Val(165)
    Broken_Bar                 : constant Character := '¦'; --Character'Val(166)
    Section_Sign               : constant Character := '§'; --Character'Val(167)
    Diaeresis                  : constant Character := '¨'; --Character'Val(168)
    Copyright_Sign             : constant Character := '©'; --Character'Val(169)
    Feminine_Ordinal_Indicator : constant Character := 'ª'; --Character'Val(170)
    Left_Angle_Quotation       : constant Character := '«'; --Character'Val(171)
    Not_Sign                   : constant Character := '¬'; --Character'Val(172)
    Soft_Hyphen                : constant Character := '-'; --Character'Val(173)
    Registered_Trade_Mark_Sign : constant Character := '®'; --Character'Val(174)
    Macron                     : constant Character := '¯'; --Character'Val(175)
```

22
```
-- Character positions 176 (16#B0#) .. 191 (16#BF#):
    Degree_Sign                : constant Character := '°'; --Character'Val(176)
    Ring_Above                 : Character renames Degree_Sign;
    Plus_Minus_Sign            : constant Character := '±'; --Character'Val(177)
    Superscript_Two            : constant Character := '²'; --Character'Val(178)
    Superscript_Three          : constant Character := '³'; --Character'Val(179)
    Acute                      : constant Character := '´'; --Character'Val(180)
    Micro_Sign                 : constant Character := 'µ'; --Character'Val(181)
    Pilcrow_Sign               : constant Character := '¶'; --Character'Val(182)
    Paragraph_Sign             : Character renames Pilcrow_Sign;
    Middle_Dot                 : constant Character := '·'; --Character'Val(183)
    Cedilla                    : constant Character := '¸'; --Character'Val(184)
    Superscript_One            : constant Character := '¹'; --Character'Val(185)
    Masculine_Ordinal_Indicator: constant Character := 'º'; --Character'Val(186)
    Right_Angle_Quotation      : constant Character := '»'; --Character'Val(187)
    Fraction_One_Quarter       : constant Character := '¼'; --Character'Val(188)
    Fraction_One_Half          : constant Character := '½'; --Character'Val(189)
    Fraction_Three_Quarters    : constant Character := '¾'; --Character'Val(190)
    Inverted_Question          : constant Character := '¿'; --Character'Val(191)
```

23
```
-- Character positions 192 (16#C0#) .. 207 (16#CF#):
    UC_A_Grave                 : constant Character := 'À'; --Character'Val(192)
    UC_A_Acute                 : constant Character := 'Á'; --Character'Val(193)
    UC_A_Circumflex            : constant Character := 'Â'; --Character'Val(194)
    UC_A_Tilde                 : constant Character := 'Ã'; --Character'Val(195)
    UC_A_Diaeresis             : constant Character := 'Ä'; --Character'Val(196)
    UC_A_Ring                  : constant Character := 'Å'; --Character'Val(197)
    UC_AE_Diphthong            : constant Character := 'Æ'; --Character'Val(198)
    UC_C_Cedilla               : constant Character := 'Ç'; --Character'Val(199)
    UC_E_Grave                 : constant Character := 'È'; --Character'Val(200)
    UC_E_Acute                 : constant Character := 'É'; --Character'Val(201)
    UC_E_Circumflex            : constant Character := 'Ê'; --Character'Val(202)
    UC_E_Diaeresis             : constant Character := 'Ë'; --Character'Val(203)
    UC_I_Grave                 : constant Character := 'Ì'; --Character'Val(204)
    UC_I_Acute                 : constant Character := 'Í'; --Character'Val(205)
    UC_I_Circumflex            : constant Character := 'Î'; --Character'Val(206)
    UC_I_Diaeresis             : constant Character := 'Ï'; --Character'Val(207)
```

24
```
-- Character positions 208 (16#D0#) .. 223 (16#DF#):
    UC_Icelandic_Eth           : constant Character := 'Ð'; --Character'Val(208)
    UC_N_Tilde                 : constant Character := 'Ñ'; --Character'Val(209)
    UC_O_Grave                 : constant Character := 'Ò'; --Character'Val(210)
    UC_O_Acute                 : constant Character := 'Ó'; --Character'Val(211)
    UC_O_Circumflex            : constant Character := 'Ô'; --Character'Val(212)
    UC_O_Tilde                 : constant Character := 'Õ'; --Character'Val(213)
    UC_O_Diaeresis             : constant Character := 'Ö'; --Character'Val(214)
    Multiplication_Sign        : constant Character := '×'; --Character'Val(215)
    UC_O_Oblique_Stroke        : constant Character := 'Ø'; --Character'Val(216)
    UC_U_Grave                 : constant Character := 'Ù'; --Character'Val(217)
    UC_U_Acute                 : constant Character := 'Ú'; --Character'Val(218)
    UC_U_Circumflex            : constant Character := 'Û'; --Character'Val(219)
    UC_U_Diaeresis             : constant Character := 'Ü'; --Character'Val(220)
    UC_Y_Acute                 : constant Character := 'Ý'; --Character'Val(221)
    UC_Icelandic_Thorn         : constant Character := 'Þ'; --Character'Val(222)
    LC_German_Sharp_S          : constant Character := 'ß'; --Character'Val(223)
```

25
```
-- Character positions 224 (16#E0#) .. 239 (16#EF#):
    LC_A_Grave                 : constant Character := 'à'; --Character'Val(224)
    LC_A_Acute                 : constant Character := 'á'; --Character'Val(225)
    LC_A_Circumflex            : constant Character := 'â'; --Character'Val(226)
    LC_A_Tilde                 : constant Character := 'ã'; --Character'Val(227)
    LC_A_Diaeresis             : constant Character := 'ä'; --Character'Val(228)
    LC_A_Ring                  : constant Character := 'å'; --Character'Val(229)
    LC_AE_Diphthong            : constant Character := 'æ'; --Character'Val(230)
    LC_C_Cedilla               : constant Character := 'ç'; --Character'Val(231)
    LC_E_Grave                 : constant Character := 'è'; --Character'Val(232)
    LC_E_Acute                 : constant Character := 'é'; --Character'Val(233)
    LC_E_Circumflex            : constant Character := 'ê'; --Character'Val(234)
    LC_E_Diaeresis             : constant Character := 'ë'; --Character'Val(235)
    LC_I_Grave                 : constant Character := 'ì'; --Character'Val(236)
    LC_I_Acute                 : constant Character := 'í'; --Character'Val(237)
    LC_I_Circumflex            : constant Character := 'î'; --Character'Val(238)
    LC_I_Diaeresis             : constant Character := 'ï'; --Character'Val(239)
```

26
```
-- Character positions 240 (16#F0#) .. 255 (16#FF#):
    LC_Icelandic_Eth           : constant Character := 'ð'; --Character'Val(240)
    LC_N_Tilde                 : constant Character := 'ñ'; --Character'Val(241)
    LC_O_Grave                 : constant Character := 'ò'; --Character'Val(242)
    LC_O_Acute                 : constant Character := 'ó'; --Character'Val(243)
    LC_O_Circumflex            : constant Character := 'ô'; --Character'Val(244)
    LC_O_Tilde                 : constant Character := 'õ'; --Character'Val(245)
    LC_O_Diaeresis             : constant Character := 'ö'; --Character'Val(246)
    Division_Sign              : constant Character := '÷'; --Character'Val(247)
    LC_O_Oblique_Stroke        : constant Character := 'ø'; --Character'Val(248)
    LC_U_Grave                 : constant Character := 'ù'; --Character'Val(249)
    LC_U_Acute                 : constant Character := 'ú'; --Character'Val(250)
    LC_U_Circumflex            : constant Character := 'û'; --Character'Val(251)
    LC_U_Diaeresis             : constant Character := 'ü'; --Character'Val(252)
    LC_Y_Acute                 : constant Character := 'ý'; --Character'Val(253)
    LC_Icelandic_Thorn         : constant Character := 'þ'; --Character'Val(254)
    LC_Y_Diaeresis             : constant Character := 'ÿ'; --Character'Val(255)
end Ada.Characters.Latin_1;
```

Implementation Permissions

27 An implementation may provide additional packages as children of Ada.Characters, to declare names for the symbols of the local character set or other character sets.

A.3.4 The Package Characters.Conversions

Static Semantics

1/2 {*AI95-00395-01*} The library package Characters.Conversions has the following declaration:

```
2/2    package Ada.Characters.Conversions is
          pragma Pure(Conversions);

3/2       function Is_Character (Item : in Wide_Character)      return Boolean;
          function Is_String    (Item : in Wide_String)         return Boolean;
          function Is_Character (Item : in Wide_Wide_Character) return Boolean;
          function Is_String    (Item : in Wide_Wide_String)    return Boolean;
          function Is_Wide_Character (Item : in Wide_Wide_Character)
             return Boolean;
          function Is_Wide_String    (Item : in Wide_Wide_String)
             return Boolean;

4/2       function To_Wide_Character (Item : in Character) return Wide_Character;
          function To_Wide_String    (Item : in String)    return Wide_String;
          function To_Wide_Wide_Character (Item : in Character)
             return Wide_Wide_Character;
          function To_Wide_Wide_String    (Item : in String)
             return Wide_Wide_String;
          function To_Wide_Wide_Character (Item : in Wide_Character)
             return Wide_Wide_Character;
          function To_Wide_Wide_String    (Item : in Wide_String)
             return Wide_Wide_String;

5/2       function To_Character (Item       : in Wide_Character;
                                Substitute : in Character := ' ')
             return Character;
          function To_String    (Item       : in Wide_String;
                                 Substitute : in Character := ' ')
             return String;
          function To_Character (Item :       in Wide_Wide_Character;
                                 Substitute : in Character := ' ')
             return Character;
          function To_String    (Item :       in Wide_Wide_String;
                                 Substitute : in Character := ' ')
             return String;
          function To_Wide_Character (Item :       in Wide_Wide_Character;
                                      Substitute : in Wide_Character := ' ')
             return Wide_Character;
          function To_Wide_String    (Item :       in Wide_Wide_String;
                                      Substitute : in Wide_Character := ' ')
             return Wide_String;

6/2    end Ada.Characters.Conversions;
```

7/2 {*AI95-00395-01*} The functions in package Characters.Conversions test Wide_Wide_Character or Wide_Character values for membership in Wide_Character or Character, or convert between corresponding characters of Wide_Wide_Character, Wide_Character, and Character.

```
8/2    function Is_Character (Item : in Wide_Character) return Boolean;
```

9/2 {*AI95-00395-01*} Returns True if Wide_Character'Pos(Item) <= Character'Pos(Character'Last).

```
10/2   function Is_Character (Item : in Wide_Wide_Character) return Boolean;
```

11/2 {*AI95-00395-01*} Returns True if Wide_Wide_Character'Pos(Item) <= Character'Pos(Character'Last).

```
12/2   function Is_Wide_Character (Item : in Wide_Wide_Character) return Boolean;
```

13/2 {*AI95-00395-01*} Returns True if Wide_Wide_Character'Pos(Item) <= Wide_Character'Pos(Wide_Character'Last).

14/2
```
function Is_String (Item : in Wide_String)      return Boolean;
function Is_String (Item : in Wide_Wide_String) return Boolean;
```

15/2 {*AI95-00395-01*} Returns True if Is_Character(Item(I)) is True for each I in Item'Range.

16/2
```
function Is_Wide_String (Item : in Wide_Wide_String) return Boolean;
```

17/2 {*AI95-00395-01*} Returns True if Is_Wide_Character(Item(I)) is True for each I in Item'Range.

18/2
```
function To_Character (Item :       in Wide_Character;
                       Substitute : in Character := ' ') return Character;
function To_Character (Item :       in Wide_Wide_Character;
                       Substitute : in Character := ' ') return Character;
```

19/2 {*AI95-00395-01*} Returns the Character corresponding to Item if Is_Character(Item), and returns the Substitute Character otherwise.

20/2
```
function To_Wide_Character (Item : in Character) return Wide_Character;
```

21/2 {*AI95-00395-01*} Returns the Wide_Character X such that Character'Pos(Item) = Wide_Character'Pos (X).

22/2
```
function To_Wide_Character (Item :       in Wide_Wide_Character;
                            Substitute : in Wide_Character := ' ')
   return Wide_Character;
```

23/2 {*AI95-00395-01*} Returns the Wide_Character corresponding to Item if Is_Wide_Character(Item), and returns the Substitute Wide_Character otherwise.

24/2
```
function To_Wide_Wide_Character (Item : in Character)
   return Wide_Wide_Character;
```

25/2 {*AI95-00395-01*} Returns the Wide_Wide_Character X such that Character'Pos(Item) = Wide_Wide_Character'Pos (X).

26/2
```
function To_Wide_Wide_Character (Item : in Wide_Character)
   return Wide_Wide_Character;
```

27/2 {*AI95-00395-01*} Returns the Wide_Wide_Character X such that Wide_Character'Pos(Item) = Wide_Wide_Character'Pos (X).

28/2
```
function To_String (Item :       in Wide_String;
                    Substitute : in Character := ' ') return String;
function To_String (Item :       in Wide_Wide_String;
                    Substitute : in Character := ' ') return String;
```

29/2 {*AI95-00395-01*} Returns the String whose range is 1..Item'Length and each of whose elements is given by To_Character of the corresponding element in Item.

30/2
```
function To_Wide_String (Item : in String) return Wide_String;
```

31/2 {*AI95-00395-01*} Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item.

32/2
```
function To_Wide_String (Item :       in Wide_Wide_String;
                         Substitute : in Wide_Character := ' ')
   return Wide_String;
```

33/2 {*AI95-00395-01*} Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item with the given Substitute Wide_Character.

34/2
```
function To_Wide_Wide_String (Item : in String) return Wide_Wide_String;
function To_Wide_Wide_String (Item : in Wide_String)
   return Wide_Wide_String;
```

35/2 {*AI95-00395-01*} Returns the Wide_Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Wide_Character of the corresponding element in Item.

Extensions to Ada 95

35.a/2 {*AI95-00395-01*} {*extensions to Ada 95*} The package Characters.Conversions is new, replacing functions previously found in Characters.Handling.

A.4 String Handling

1/2 {*AI95-00285-01*} This clause presents the specifications of the package Strings and several child packages, which provide facilities for dealing with string data. Fixed-length, bounded-length, and unbounded-length strings are supported, for String, Wide_String, and Wide_Wide_String. The string-handling subprograms include searches for pattern strings and for characters in program-specified sets, translation (via a character-to-character mapping), and transformation (replacing, inserting, overwriting, and deleting of substrings).

Extensions to Ada 83

1.a {*extensions to Ada 83*} This clause is new to Ada 95.

Wording Changes from Ada 95

1.b/2 {*AI95-00285-01*} Included Wide_Wide_String in this description; the individual changes are documented as extensions as needed.

A.4.1 The Package Strings

1 The package Strings provides declarations common to the string handling packages.

Static Semantics

2 The library package Strings has the following declaration:

3
```
package Ada.Strings is
   pragma Pure(Strings);
```

4/2
```
{AI95-00285-01}    Space      : constant Character      := ' ';
   Wide_Space : constant Wide_Character := ' ';
   Wide_Wide_Space : constant Wide_Wide_Character := ' ';
```

5
```
   Length_Error, Pattern_Error, Index_Error, Translation_Error : exception;
```

6
```
   type Alignment  is (Left, Right, Center);
   type Truncation is (Left, Right, Error);
   type Membership is (Inside, Outside);
   type Direction  is (Forward, Backward);
   type Trim_End   is (Left, Right, Both);
end Ada.Strings;
```

Incompatibilities With Ada 95

6.a/2 {*AI95-00285-01*} {*incompatibilities with Ada 95*} Constant Wide_Wide_Space is newly added to Ada.Strings. If Ada.Strings is referenced in a use_clause, and an entity *E* with a defining_identifier of Wide_Wide_Space is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

A.4.2 The Package Strings.Maps

The package Strings.Maps defines the types, operations, and other entities needed for character sets and character-to-character mappings. 1

Static Semantics

The library package Strings.Maps has the following declaration: 2

```
{AI95-00362-01} package Ada.Strings.Maps is                                    3/2
   pragma Pure(Maps);

{AI95-00161-01}     -- Representation for a set of character values:            4/2
   type Character_Set is private;
   pragma Preelaborable_Initialization(Character_Set);

   Null_Set : constant Character_Set;                                           5

   type Character_Range is                                                      6
     record
        Low  : Character;
        High : Character;
     end record;
   --  Represents Character range Low..High

   type Character_Ranges is array (Positive range <>) of Character_Range;       7

   function To_Set    (Ranges : in Character_Ranges)return Character_Set;       8

   function To_Set    (Span   : in Character_Range)return Character_Set;        9

   function To_Ranges (Set    : in Character_Set)  return Character_Ranges;     10

   function "="   (Left, Right : in Character_Set) return Boolean;              11

   function "not" (Right : in Character_Set)       return Character_Set;        12
   function "and" (Left, Right : in Character_Set) return Character_Set;
   function "or"  (Left, Right : in Character_Set) return Character_Set;
   function "xor" (Left, Right : in Character_Set) return Character_Set;
   function "-"   (Left, Right : in Character_Set) return Character_Set;

   function Is_In (Element : in Character;                                      13
                   Set     : in Character_Set)
      return Boolean;

   function Is_Subset (Elements : in Character_Set;                             14
                       Set      : in Character_Set)
      return Boolean;

   function "<=" (Left  : in Character_Set;                                     15
                  Right : in Character_Set)
      return Boolean renames Is_Subset;

   -- Alternative representation for a set of character values:                16
   subtype Character_Sequence is String;

   function To_Set (Sequence  : in Character_Sequence)return Character_Set;     17

   function To_Set (Singleton : in Character)     return Character_Set;         18

   function To_Sequence (Set  : in Character_Set) return Character_Sequence;    19

{AI95-00161-01}     -- Representation for a character to character mapping:    20/2
   type Character_Mapping is private;
   pragma Preelaborable_Initialization(Character_Mapping);

   function Value (Map     : in Character_Mapping;                              21
                   Element : in Character)
      return Character;

   Identity : constant Character_Mapping;                                       22

   function To_Mapping (From, To : in Character_Sequence)                       23
      return Character_Mapping;
```

24
```
      function To_Domain (Map : in Character_Mapping)
         return Character_Sequence;
      function To_Range  (Map : in Character_Mapping)
         return Character_Sequence;
```
25
```
      type Character_Mapping_Function is
         access function (From : in Character) return Character;
```
26
```
   private
      ... -- not specified by the language
   end Ada.Strings.Maps;
```
27 An object of type Character_Set represents a set of characters.

28 Null_Set represents the set containing no characters.

29 An object Obj of type Character_Range represents the set of characters in the range Obj.Low .. Obj.High.

30 An object Obj of type Character_Ranges represents the union of the sets corresponding to Obj(I) for I in Obj'Range.

31
```
   function To_Set (Ranges : in Character_Ranges) return Character_Set;
```
32 If Ranges'Length=0 then Null_Set is returned; otherwise the returned value represents the set corresponding to Ranges.

33
```
   function To_Set (Span : in Character_Range) return Character_Set;
```
34 The returned value represents the set containing each character in Span.

35
```
   function To_Ranges (Set : in Character_Set) return Character_Ranges;
```
36 If Set = Null_Set then an empty Character_Ranges array is returned; otherwise the shortest array of contiguous ranges of Character values in Set, in increasing order of Low, is returned.

37
```
   function "=" (Left, Right : in Character_Set) return Boolean;
```
38 The function "=" returns True if Left and Right represent identical sets, and False otherwise.

39 Each of the logical operators "**not**", "**and**", "**or**", and "**xor**" returns a Character_Set value that represents the set obtained by applying the corresponding operation to the set(s) represented by the parameter(s) of the operator. "–"(Left, Right) is equivalent to "and"(Left, "not"(Right)).

39.a **Reason:** The set minus operator is provided for efficiency.

40
```
   function Is_In (Element : in Character;
                   Set     : in Character_Set);
      return Boolean;
```
41 Is_In returns True if Element is in Set, and False otherwise.

42
```
   function Is_Subset (Elements : in Character_Set;
                       Set      : in Character_Set)
      return Boolean;
```
43 Is_Subset returns True if Elements is a subset of Set, and False otherwise.

44
```
   subtype Character_Sequence is String;
```
45 The Character_Sequence subtype is used to portray a set of character values and also to identify the domain and range of a character mapping.

45.a **Reason:** Although a named subtype is redundant — the predefined type String could have been used for the parameter to To_Set and To_Mapping below — the use of a differently named subtype identifies the intended purpose of the parameter.

46
```
function To_Set (Sequence  : in Character_Sequence) return Character_Set;

function To_Set (Singleton : in Character)          return Character_Set;
```

47 Sequence portrays the set of character values that it explicitly contains (ignoring duplicates). Singleton portrays the set comprising a single Character. Each of the To_Set functions returns a Character_Set value that represents the set portrayed by Sequence or Singleton.

48
```
function To_Sequence (Set : in Character_Set) return Character_Sequence;
```

49 The function To_Sequence returns a Character_Sequence value containing each of the characters in the set represented by Set, in ascending order with no duplicates.

50
```
type Character_Mapping is private;
```

51 An object of type Character_Mapping represents a Character-to-Character mapping.

52
```
function Value (Map     : in Character_Mapping;
                Element : in Character)
   return Character;
```

53 The function Value returns the Character value to which Element maps with respect to the mapping represented by Map.

54 {*match (a character to a pattern character)*} A character C *matches* a pattern character P with respect to a given Character_Mapping value Map if Value(Map, C) = P. {*match (a string to a pattern string)*} A string S *matches* a pattern string P with respect to a given Character_Mapping if their lengths are the same and if each character in S matches its corresponding character in the pattern string P.

54.a **Discussion:** In an earlier version of the string handling packages, the definition of matching was symmetrical, namely C matches P if Value(Map,C) = Value(Map,P). However, applying the mapping to the pattern was confusing according to some reviewers. Furthermore, if the symmetrical version is needed, it can be achieved by applying the mapping to the pattern (via translation) prior to passing it as a parameter.

55 String handling subprograms that deal with character mappings have parameters whose type is Character_Mapping.

56
```
Identity : constant Character_Mapping;
```

57 Identity maps each Character to itself.

58
```
function To_Mapping (From, To : in Character_Sequence)
    return Character_Mapping;
```

59 To_Mapping produces a Character_Mapping such that each element of From maps to the corresponding element of To, and each other character maps to itself. If From'Length /= To'Length, or if some character is repeated in From, then Translation_Error is propagated.

60
```
function To_Domain (Map : in Character_Mapping) return Character_Sequence;
```

61 To_Domain returns the shortest Character_Sequence value D such that each character not in D maps to itself, and such that the characters in D are in ascending order. The lower bound of D is 1.

62
```
function To_Range  (Map : in Character_Mapping) return Character_Sequence;
```

63/1 {*8652/0048*} {*AI95-00151-01*} To_Range returns the Character_Sequence value R, such that if D = To_Domain(Map), then R has the same bounds as D, and D(I) maps to R(I) for each I in D'Range.

64 An object F of type Character_Mapping_Function maps a Character value C to the Character value F.**all**(C), which is said to *match* C with respect to mapping function F. {*match (a character to a pattern character, with respect to a character mapping function)*}

NOTES

65 7 Character_Mapping and Character_Mapping_Function are used both for character equivalence mappings in the search subprograms (such as for case insensitivity) and as transformational mappings in the Translate subprograms.

66 8 To_Domain(Identity) and To_Range(Identity) each returns the null string.

66.a **Reason:** Package Strings.Maps is not pure, since it declares an access-to-subprogram type.

Examples

67 To_Mapping("ABCD", "ZZAB") returns a Character_Mapping that maps 'A' and 'B' to 'Z', 'C' to 'A', 'D' to 'B', and each other Character to itself.

Extensions to Ada 95

67.a/2 {*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Added pragma Preelaborable_Initialization to types Character_Set and Character_Mapping, so that they can be used to declare default-initialized objects in preelaborated units.

67.b/2 {*AI95-00362-01*} Strings.Maps is now Pure, so it can be used in pure units.

Wording Changes from Ada 95

67.c/2 {*8652/0048*} {*AI95-00151-01*} **Corrigendum:** Corrected the definition of the range of the result of To_Range, since the Ada 95 definition makes no sense.

A.4.3 Fixed-Length String Handling

1 The language-defined package Strings.Fixed provides string-handling subprograms for fixed-length strings; that is, for values of type Standard.String. Several of these subprograms are procedures that modify the contents of a String that is passed as an **out** or an **in out** parameter; each has additional parameters to control the effect when the logical length of the result differs from the parameter's length.

2 For each function that returns a String, the lower bound of the returned value is 1.

2.a/2 **Discussion:** {*AI95-00114-01*} Most operations that yield a String are provided both as a function and as a procedure. The functional form is possibly a more aesthetic style but may introduce overhead due to extra copying or dynamic memory usage in some implementations. Thus a procedural form, with an **in out** parameter so that all copying is done `in place', is also supplied.

3 The basic model embodied in the package is that a fixed-length string comprises significant characters and possibly padding (with space characters) on either or both ends. When a shorter string is copied to a longer string, padding is inserted, and when a longer string is copied to a shorter one, padding is stripped. The Move procedure in Strings.Fixed, which takes a String as an **out** parameter, allows the programmer to control these effects. Similar control is provided by the string transformation procedures.

Static Semantics

4 The library package Strings.Fixed has the following declaration:

5
```
with Ada.Strings.Maps;
package Ada.Strings.Fixed is
   pragma Preelaborate(Fixed);
```

6
```
-- "Copy" procedure for strings of possibly different lengths
```

7
```
   procedure Move (Source  : in  String;
                   Target  : out String;
                   Drop    : in  Truncation := Error;
                   Justify : in  Alignment  := Left;
                   Pad     : in  Character  := Space);
```

8
```
-- Search subprograms
```

8.1/2 {*AI95-00301-01*}
```
   function Index (Source  : in String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping := Maps.Identity)
      return Natural;
```

8.2/2 {*AI95-00301-01*}
```
   function Index (Source  : in String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping_Function)
      return Natural;
```

9
```
   function Index (Source   : in String;
                   Pattern  : in String;
                   Going    : in Direction := Forward;
                   Mapping  : in Maps.Character_Mapping
                                 := Maps.Identity)
      return Natural;
```

10
```
   function Index (Source   : in String;
                   Pattern  : in String;
                   Going    : in Direction := Forward;
                   Mapping  : in Maps.Character_Mapping_Function)
      return Natural;
```

10.1/2 {*AI95-00301-01*}
```
   function Index (Source  : in String;
                   Set     : in Maps.Character_Set;
                   From    : in Positive;
                   Test    : in Membership := Inside;
                   Going   : in Direction := Forward)
      return Natural;
```

11
```
   function Index (Source : in String;
                   Set    : in Maps.Character_Set;
                   Test   : in Membership := Inside;
                   Going  : in Direction  := Forward)
      return Natural;
```

11.1/2 {*AI95-00301-01*}
```
   function Index_Non_Blank (Source : in String;
                             From   : in Positive;
                             Going  : in Direction := Forward)
      return Natural;
```

12
```
   function Index_Non_Blank (Source : in String;
                             Going  : in Direction := Forward)
      return Natural;
```

13
```
   function Count (Source   : in String;
                   Pattern  : in String;
                   Mapping  : in Maps.Character_Mapping
                                  := Maps.Identity)
      return Natural;
```

14
```
   function Count (Source   : in String;
                   Pattern  : in String;
                   Mapping  : in Maps.Character_Mapping_Function)
      return Natural;
```

15
```
   function Count (Source   : in String;
                   Set      : in Maps.Character_Set)
      return Natural;
```

16
```
   procedure Find_Token (Source : in String;
                         Set    : in Maps.Character_Set;
                         Test   : in Membership;
                         First  : out Positive;
                         Last   : out Natural);
```

17 -- *String translation subprograms*

18
```
   function Translate (Source  : in String;
                       Mapping : in Maps.Character_Mapping)
      return String;
```

19
```
   procedure Translate (Source  : in out String;
                        Mapping : in Maps.Character_Mapping);
```

20
```
   function Translate (Source  : in String;
                       Mapping : in Maps.Character_Mapping_Function)
      return String;
```

21
```
   procedure Translate (Source  : in out String;
                        Mapping : in Maps.Character_Mapping_Function);
```

22 -- *String transformation subprograms*

23
```
   function Replace_Slice (Source   : in String;
                           Low      : in Positive;
                           High     : in Natural;
                           By       : in String)
      return String;
```

24
```
   procedure Replace_Slice (Source   : in out String;
                            Low      : in Positive;
                            High     : in Natural;
                            By       : in String;
                            Drop     : in Truncation := Error;
                            Justify  : in Alignment  := Left;
                            Pad      : in Character  := Space);
```

25
```
   function Insert (Source   : in String;
                    Before   : in Positive;
                    New_Item : in String)
      return String;
```

26
```
   procedure Insert (Source   : in out String;
                     Before   : in Positive;
                     New_Item : in String;
                     Drop     : in Truncation := Error);
```

27
```
   function Overwrite (Source   : in String;
                       Position : in Positive;
                       New_Item : in String)
      return String;
```

28
```
   procedure Overwrite (Source   : in out String;
                        Position : in Positive;
                        New_Item : in String;
                        Drop     : in Truncation := Right);
```

29
```
   function Delete (Source  : in String;
                    From    : in Positive;
                    Through : in Natural)
      return String;
```

30
```
   procedure Delete (Source  : in out String;
                     From    : in Positive;
                     Through : in Natural;
                     Justify : in Alignment := Left;
                     Pad     : in Character := Space);
```

31 --*String selector subprograms*
```
   function Trim (Source : in String;
                  Side   : in Trim_End)
      return String;
```

32
```
   procedure Trim (Source  : in out String;
                   Side    : in Trim_End;
                   Justify : in Alignment := Left;
                   Pad     : in Character := Space);
   function Trim (Source : in String;                                      33
                  Left   : in Maps.Character_Set;
                  Right  : in Maps.Character_Set)
      return String;
   procedure Trim (Source  : in out String;                                34
                   Left    : in Maps.Character_Set;
                   Right   : in Maps.Character_Set;
                   Justify : in Alignment := Strings.Left;
                   Pad     : in Character := Space);
   function Head (Source : in String;                                      35
                  Count  : in Natural;
                  Pad    : in Character := Space)
      return String;
   procedure Head (Source  : in out String;                                36
                   Count   : in Natural;
                   Justify : in Alignment := Left;
                   Pad     : in Character := Space);
   function Tail (Source : in String;                                      37
                  Count  : in Natural;
                  Pad    : in Character := Space)
      return String;
   procedure Tail (Source  : in out String;                                38
                   Count   : in Natural;
                   Justify : in Alignment := Left;
                   Pad     : in Character := Space);
--String constructor functions                                              39
   function "*" (Left  : in Natural;                                       40
                 Right : in Character) return String;
   function "*" (Left  : in Natural;                                       41
                 Right : in String) return String;
end Ada.Strings.Fixed;                                                      42
```

The effects of the above subprograms are as follows. 43

```
procedure Move (Source  : in  String;                                      44
                Target  : out String;
                Drop    : in  Truncation := Error;
                Justify : in  Alignment  := Left;
                Pad     : in  Character  := Space);
```

45 The Move procedure copies characters from Source to Target. If Source has the same length as Target, then the effect is to assign Source to Target. If Source is shorter than Target then:

- 46 If Justify=Left, then Source is copied into the first Source'Length characters of Target.
- 47 If Justify=Right, then Source is copied into the last Source'Length characters of Target.
- 48 If Justify=Center, then Source is copied into the middle Source'Length characters of Target. In this case, if the difference in length between Target and Source is odd, then the extra Pad character is on the right.
- 49 Pad is copied to each Target character not otherwise assigned.

50 If Source is longer than Target, then the effect is based on Drop.

- 51 If Drop=Left, then the rightmost Target'Length characters of Source are copied into Target.
- 52 If Drop=Right, then the leftmost Target'Length characters of Source are copied into Target.
- 53 If Drop=Error, then the effect depends on the value of the Justify parameter and also on whether any characters in Source other than Pad would fail to be copied:

54 • If Justify=Left, and if each of the rightmost Source'Length-Target'Length characters in Source is Pad, then the leftmost Target'Length characters of Source are copied to Target.

55 • If Justify=Right, and if each of the leftmost Source'Length-Target'Length characters in Source is Pad, then the rightmost Target'Length characters of Source are copied to Target.

56 • Otherwise, Length_Error is propagated.

56.a **Ramification:** The Move procedure will work even if Source and Target overlap.

56.b **Reason:** The order of parameters (Source before Target) corresponds to the order in COBOL's MOVE verb.

56.1/2
```
function Index (Source  : in String;
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping := Maps.Identity)
   return Natural;

function Index (Source  : in String;
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping_Function)
   return Natural;
```

56.2/2 {*AI95-00301-01*} Each Index function searches, starting from From, for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If From is not in Source'Range, then Index_Error is propagated. If Going = Forward, then Index returns the smallest index I which is greater than or equal to From such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern and has an upper bound less than or equal to From. If there is no such slice, then 0 is returned. If Pattern is the null string, then Pattern_Error is propagated.

56.c/2 **Discussion:** There is no default parameter for From; the default value would need to depend on other parameters (the bounds of Source and the direction Going). It is better to use overloaded functions rather than a special value to represent the default.

56.d/2 There is no default value for the Mapping parameter that is a Character_Mapping_Function; if there were, a call would be ambiguous since there is also a default for the Mapping parameter that is a Character_Mapping.

57
```
function Index (Source   : in String;
                Pattern  : in String;
                Going    : in Direction := Forward;
                Mapping  : in Maps.Character_Mapping
                              := Maps.Identity)
   return Natural;

function Index (Source   : in String;
                Pattern  : in String;
                Going    : in Direction := Forward;
                Mapping  : in Maps.Character_Mapping_Function)
   return Natural;
```

58/2 {*AI95-00301-01*} If Going = Forward, returns

58.1/2
```
   Index (Source, Pattern, Source'First, Forward, Mapping);
```

58.2/2 otherwise returns

58.3/2
```
   Index (Source, Pattern, Source'Last, Backward, Mapping);
```

58.a/2 *This paragraph was deleted.*~~There is no default value for the Mapping parameter that is a Character_Mapping_Function; if there were, a call would be ambiguous since there is also a default for the Mapping parameter that is a Character_Mapping.~~

58.4/2
```
function Index (Source  : in String;
                Set     : in Maps.Character_Set;
                From    : in Positive;
                Test    : in Membership := Inside;
                Going   : in Direction := Forward)
   return Natural;
```

58.5/2 {*AI95-00301-01*} Index searches for the first or last occurrence of any of a set of characters (when Test=Inside), or any of the complement of a set of characters (when Test=Outside). If From is not in Source'Range, then Index_Error is propagated. Otherwise, it returns the smallest index I >= From (if Going=Forward) or the largest index I <= From (if Going=Backward) such that Source(I) satisfies the Test condition with respect to Set; it returns 0 if there is no such Character in Source.

59
```
function Index (Source : in String;
                Set    : in Maps.Character_Set;
                Test   : in Membership := Inside;
                Going  : in Direction  := Forward)
   return Natural;
```

60/2 {*AI95-00301-01*} If Going = Forward, returns

60.1/2
```
   Index (Source, Set, Source'First, Test, Forward);
```

60.2/2 otherwise returns

60.3/2
```
   Index (Source, Set, Source'Last, Test, Backward);
```

60.4/2
```
function Index_Non_Blank (Source : in String;
                          From   : in Positive;
                          Going  : in Direction := Forward)
   return Natural;
```

60.5/2 {*AI95-00301-01*} Returns Index (Source, Maps.To_Set(Space), From, Outside, Going);

61
```
function Index_Non_Blank (Source : in String;
                          Going  : in Direction := Forward)
   return Natural;
```

62 Returns Index(Source, Maps.To_Set(Space), Outside, Going)

63
```
function Count (Source   : in String;
                Pattern  : in String;
                Mapping  : in Maps.Character_Mapping
                             := Maps.Identity)
   return Natural;

function Count (Source   : in String;
                Pattern  : in String;
                Mapping  : in Maps.Character_Mapping_Function)
   return Natural;
```

64 Returns the maximum number of nonoverlapping slices of Source that match Pattern with respect to Mapping. If Pattern is the null string then Pattern_Error is propagated.

64.a **Reason:** We say `maximum number' because it is possible to slice a source string in different ways yielding different numbers of matches. For example if Source is "ABABABA" and Pattern is "ABA", then Count yields 2, although there is a partitioning of Source that yields just 1 match, for the middle slice. Saying `maximum number' is equivalent to saying that the pattern match starts either at the low index or the high index position.

65
```
function Count (Source   : in String;
                Set      : in Maps.Character_Set)
   return Natural;
```

66 Returns the number of occurrences in Source of characters that are in Set.

67
```
procedure Find_Token (Source : in String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

68/1 {*8652/0049*} {*AI95-00128-01*} Find_Token returns in First and Last the indices of the beginning and end of the first slice of Source all of whose elements satisfy the Test condition, and such that the elements (if any) immediately before and after the slice do not satisfy the Test condition. If no such slice exists, then the value returned for Last is zero, and the value returned for First is Source'First; however, if Source'First is not in Positive then Constraint_Error {*Constraint_Error (raised by failure of run-time check)*} is raised.

69
```
function Translate (Source  : in String;
                    Mapping : in Maps.Character_Mapping)
   return String;

function Translate (Source  : in String;
                    Mapping : in Maps.Character_Mapping_Function)
   return String;
```

70 Returns the string S whose length is Source'Length and such that S(I) is the character to which Mapping maps the corresponding element of Source, for I in 1..Source'Length.

71
```
procedure Translate (Source  : in out String;
                     Mapping : in Maps.Character_Mapping);

procedure Translate (Source  : in out String;
                     Mapping : in Maps.Character_Mapping_Function);
```

72 Equivalent to Source := Translate(Source, Mapping).

73
```
function Replace_Slice (Source   : in String;
                        Low      : in Positive;
                        High     : in Natural;
                        By       : in String)
   return String;
```

74/1 {*8652/0049*} {*AI95-00128-01*} If Low > Source'Last+1, or High < Source'First–1, then Index_Error is propagated. Otherwise:

74.1/1
- {*8652/0049*} {*AI95-00128-01*} If High >= Low, then the returned string comprises Source(Source'First..Low–1) & By & Source(High+1..Source'Last), but with lower bound 1.

74.2/1
- {*8652/0049*} {*AI95-00128-01*} If High < Low, then the returned string is Insert(Source, Before=>Low, New_Item=>By).

75
```
procedure Replace_Slice (Source   : in out String;
                         Low      : in Positive;
                         High     : in Natural;
                         By       : in String;
                         Drop     : in Truncation := Error;
                         Justify  : in Alignment  := Left;
                         Pad      : in Character  := Space);
```

76 Equivalent to Move(Replace_Slice(Source, Low, High, By), Source, Drop, Justify, Pad).

```
function Insert (Source   : in String;                                    77
                 Before   : in Positive;
                 New_Item : in String)
   return String;
```

78 Propagates Index_Error if Before is not in Source'First .. Source'Last+1; otherwise returns Source(Source'First..Before–1) & New_Item & Source(Before..Source'Last), but with lower bound 1.

```
procedure Insert (Source   : in out String;                               79
                  Before   : in Positive;
                  New_Item : in String;
                  Drop     : in Truncation := Error);
```

80 Equivalent to Move(Insert(Source, Before, New_Item), Source, Drop).

```
function Overwrite (Source   : in String;                                 81
                    Position : in Positive;
                    New_Item : in String)
   return String;
```

82 Propagates Index_Error if Position is not in Source'First .. Source'Last+1; otherwise returns the string obtained from Source by consecutively replacing characters starting at Position with corresponding characters from New_Item. If the end of Source is reached before the characters in New_Item are exhausted, the remaining characters from New_Item are appended to the string.

```
procedure Overwrite (Source   : in out String;                            83
                     Position : in Positive;
                     New_Item : in String;
                     Drop     : in Truncation := Right);
```

84 Equivalent to Move(Overwrite(Source, Position, New_Item), Source, Drop).

```
function Delete (Source  : in String;                                     85
                 From    : in Positive;
                 Through : in Natural)
   return String;
```

86/1 {*8652/0049*} {*AI95-00128-01*} If From <= Through, the returned string is Replace_Slice(Source, From, Through, ""), otherwise it is Source with lower bound 1.

```
procedure Delete (Source  : in out String;                                87
                  From    : in Positive;
                  Through : in Natural;
                  Justify : in Alignment := Left;
                  Pad     : in Character := Space);
```

88 Equivalent to Move(Delete(Source, From, Through), Source, Justify => Justify, Pad => Pad).

```
function Trim (Source : in String;                                        89
               Side   : in Trim_End)
  return String;
```

90 Returns the string obtained by removing from Source all leading Space characters (if Side = Left), all trailing Space characters (if Side = Right), or all leading and trailing Space characters (if Side = Both).

```
procedure Trim (Source  : in out String;                                  91
                Side    : in Trim_End;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

92 Equivalent to Move(Trim(Source, Side), Source, Justify=>Justify, Pad=>Pad).

93
```
function Trim (Source : in String;
               Left   : in Maps.Character_Set;
               Right  : in Maps.Character_Set)
   return String;
```

94 Returns the string obtained by removing from Source all leading characters in Left and all trailing characters in Right.

95
```
procedure Trim (Source  : in out String;
                Left    : in Maps.Character_Set;
                Right   : in Maps.Character_Set;
                Justify : in Alignment := Strings.Left;
                Pad     : in Character := Space);
```

96 Equivalent to Move(Trim(Source, Left, Right), Source, Justify => Justify, Pad=>Pad).

97
```
function Head (Source : in String;
               Count  : in Natural;
               Pad    : in Character := Space)
   return String;
```

98 Returns a string of length Count. If Count <= Source'Length, the string comprises the first Count characters of Source. Otherwise its contents are Source concatenated with Count–Source'Length Pad characters.

99
```
procedure Head (Source  : in out String;
                Count   : in Natural;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

100 Equivalent to Move(Head(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad).

101
```
function Tail (Source : in String;
               Count  : in Natural;
               Pad    : in Character := Space)
   return String;
```

102 Returns a string of length Count. If Count <= Source'Length, the string comprises the last Count characters of Source. Otherwise its contents are Count-Source'Length Pad characters concatenated with Source.

103
```
procedure Tail (Source  : in out String;
                Count   : in Natural;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

104 Equivalent to Move(Tail(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad).

105
```
function "*" (Left  : in Natural;
              Right : in Character) return String;

function "*" (Left  : in Natural;
              Right : in String) return String;
```

106/1 {*8652/0049*} {*AI95-00128-01*} These functions replicate a character or string a specified number of times. The first function returns a string whose length is Left and each of whose elements is Right. The second function returns a string whose length is Left*Right'Length and whose value is the null string if Left = 0 and otherwise is (Left–1)*Right & Right with lower bound 1.

NOTES

107 9 In the Index and Count functions taking Pattern and Mapping parameters, the actual String parameter passed to Pattern should comprise characters occurring as target characters of the mapping. Otherwise the pattern will not match.

108 10 In the Insert subprograms, inserting at the end of a string is obtained by passing Source'Last+1 as the Before parameter.

11 {*Constraint_Error (raised by failure of run-time check)*} If a null Character_Mapping_Function is passed to any of the string handling subprograms, Constraint_Error is propagated. 109

Incompatibilities With Ada 95

{*AI95-00301-01*} {*incompatibilities with Ada 95*} Overloaded versions of Index and Index_Non_Blank are newly added to Strings.Fixed. If Strings.Fixed is referenced in a use_clause, and an entity *E* with a defining_identifier of Index or Index_Non_Blank is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 109.a/2

Wording Changes from Ada 95

{*8652/0049*} {*AI95-00128-01*} **Corrigendum:** Clarified that Find_Token may raise Constraint_Error if Source'First is not in Positive (which is only possible for a null string). 109.b/2

{*8652/0049*} {*AI95-00128-01*} **Corrigendum:** Clarified that Replace_Slice, Delete, and "*" always return a string with lower bound 1. 109.c/2

A.4.4 Bounded-Length String Handling

1 The language-defined package Strings.Bounded provides a generic package each of whose instances yields a private type Bounded_String and a set of operations. An object of a particular Bounded_String type represents a String whose low bound is 1 and whose length can vary conceptually between 0 and a maximum size established at the generic instantiation. The subprograms for fixed-length string handling are either overloaded directly for Bounded_String, or are modified as needed to reflect the variability in length. Additionally, since the Bounded_String type is private, appropriate constructor and selector operations are provided.

Reason: Strings.Bounded declares an inner generic package, versus itself being directly a generic child of Strings, in order to retain compatibility with a version of the string-handling packages that is generic with respect to the character and string types. 1.a

Reason: The bound of a bounded-length string is specified as a parameter to a generic, versus as the value for a discriminant, because of the inappropriateness of assignment and equality of discriminated types for the copying and comparison of bounded strings. 1.b

Static Semantics

2 The library package Strings.Bounded has the following declaration:

```
3
with Ada.Strings.Maps;
package Ada.Strings.Bounded is
   pragma Preelaborate(Bounded);

4
   generic
      Max   : Positive;     -- Maximum length of a Bounded_String
   package Generic_Bounded_Length is

5
      Max_Length : constant Positive := Max;

6
      type Bounded_String is private;

7
      Null_Bounded_String : constant Bounded_String;

8
      subtype Length_Range is Natural range 0 .. Max_Length;

9
      function Length (Source : in Bounded_String) return Length_Range;

10
   -- Conversion, Concatenation, and Selection functions

11
      function To_Bounded_String (Source : in String;
                                  Drop   : in Truncation := Error)
         return Bounded_String;

12
      function To_String (Source : in Bounded_String) return String;

12.1/2
{AI95-00301-01}       procedure Set_Bounded_String
         (Target :    out Bounded_String;
          Source : in     String;
          Drop   : in     Truncation := Error);
```

13
```
      function Append (Left, Right : in Bounded_String;
                       Drop        : in Truncation  := Error)
         return Bounded_String;
```
14
```
      function Append (Left  : in Bounded_String;
                       Right : in String;
                       Drop  : in Truncation := Error)
         return Bounded_String;
```
15
```
      function Append (Left  : in String;
                       Right : in Bounded_String;
                       Drop  : in Truncation := Error)
         return Bounded_String;
```
16
```
      function Append (Left  : in Bounded_String;
                       Right : in Character;
                       Drop  : in Truncation := Error)
         return Bounded_String;
```
17
```
      function Append (Left  : in Character;
                       Right : in Bounded_String;
                       Drop  : in Truncation := Error)
         return Bounded_String;
```
18
```
      procedure Append (Source   : in out Bounded_String;
                        New_Item : in Bounded_String;
                        Drop     : in Truncation  := Error);
```
19
```
      procedure Append (Source   : in out Bounded_String;
                        New_Item : in String;
                        Drop     : in Truncation  := Error);
```
20
```
      procedure Append (Source   : in out Bounded_String;
                        New_Item : in Character;
                        Drop     : in Truncation  := Error);
```
21
```
      function "&" (Left, Right : in Bounded_String)
         return Bounded_String;
```
22
```
      function "&" (Left : in Bounded_String; Right : in String)
         return Bounded_String;
```
23
```
      function "&" (Left : in String; Right : in Bounded_String)
         return Bounded_String;
```
24
```
      function "&" (Left : in Bounded_String; Right : in Character)
         return Bounded_String;
```
25
```
      function "&" (Left : in Character; Right : in Bounded_String)
         return Bounded_String;
```
26
```
      function Element (Source : in Bounded_String;
                        Index  : in Positive)
         return Character;
```
27
```
      procedure Replace_Element (Source : in out Bounded_String;
                                 Index  : in Positive;
                                 By     : in Character);
```
28
```
      function Slice (Source : in Bounded_String;
                      Low    : in Positive;
                      High   : in Natural)
         return String;
```
28.1/2 {*AI95-00301-01*}
```
      function Bounded_Slice
         (Source : in Bounded_String;
          Low    : in Positive;
          High   : in Natural)
             return Bounded_String;
```
28.2/2 {*AI95-00301-01*}
```
      procedure Bounded_Slice
         (Source : in     Bounded_String;
          Target :    out Bounded_String;
          Low    : in     Positive;
          High   : in     Natural);
```

29
```
function "="  (Left, Right : in Bounded_String) return Boolean;
function "="  (Left : in Bounded_String; Right : in String)
  return Boolean;
```
30
```
function "="  (Left : in String; Right : in Bounded_String)
  return Boolean;
```
31
```
function "<"  (Left, Right : in Bounded_String) return Boolean;
```
32
```
function "<"  (Left : in Bounded_String; Right : in String)
  return Boolean;
```
33
```
function "<"  (Left : in String; Right : in Bounded_String)
  return Boolean;
```
34
```
function "<=" (Left, Right : in Bounded_String) return Boolean;
```
35
```
function "<="  (Left : in Bounded_String; Right : in String)
  return Boolean;
```
36
```
function "<="  (Left : in String; Right : in Bounded_String)
  return Boolean;
```
37
```
function ">"  (Left, Right : in Bounded_String) return Boolean;
```
38
```
function ">"  (Left : in Bounded_String; Right : in String)
  return Boolean;
```
39
```
function ">"  (Left : in String; Right : in Bounded_String)
  return Boolean;
```
40
```
function ">=" (Left, Right : in Bounded_String) return Boolean;
```
41
```
function ">="  (Left : in Bounded_String; Right : in String)
  return Boolean;
```
42
```
function ">="  (Left : in String; Right : in Bounded_String)
  return Boolean;
```
43/2
```
{AI95-00301-01}      -- Search subprograms
```
43.1/2
```
{AI95-00301-01}       function Index (Source  : in Bounded_String;
                      Pattern : in String;
                      From    : in Positive;
                      Going   : in Direction := Forward;
                      Mapping : in Maps.Character_Mapping := Maps.Identity)
   return Natural;
```
43.2/2
```
{AI95-00301-01}       function Index (Source  : in Bounded_String;
                      Pattern : in String;
                      From    : in Positive;
                      Going   : in Direction := Forward;
                      Mapping : in Maps.Character_Mapping_Function)
   return Natural;
```
44
```
function Index (Source   : in Bounded_String;
                Pattern  : in String;
                Going    : in Direction := Forward;
                Mapping  : in Maps.Character_Mapping
                           := Maps.Identity)
   return Natural;
```
45
```
function Index (Source   : in Bounded_String;
                Pattern  : in String;
                Going    : in Direction := Forward;
                Mapping  : in Maps.Character_Mapping_Function)
   return Natural;
```
45.1/2
```
{AI95-00301-01}       function Index (Source  : in Bounded_String;
                      Set     : in Maps.Character_Set;
                      From    : in Positive;
                      Test    : in Membership := Inside;
                      Going   : in Direction := Forward)
   return Natural;
```

46
```
      function Index (Source : in Bounded_String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership := Inside;
                      Going  : in Direction  := Forward)
         return Natural;
```

46.1/2 {*AI95-00301-01*}
```
      function Index_Non_Blank (Source : in Bounded_String;
                                From   : in Positive;
                                Going  : in Direction := Forward)
         return Natural;
```

47
```
      function Index_Non_Blank (Source : in Bounded_String;
                                Going  : in Direction := Forward)
         return Natural;
```

48
```
      function Count (Source   : in Bounded_String;
                      Pattern  : in String;
                      Mapping  : in Maps.Character_Mapping
                                   := Maps.Identity)
         return Natural;
```

49
```
      function Count (Source   : in Bounded_String;
                      Pattern  : in String;
                      Mapping  : in Maps.Character_Mapping_Function)
         return Natural;
```

50
```
      function Count (Source   : in Bounded_String;
                      Set      : in Maps.Character_Set)
         return Natural;
```

51
```
      procedure Find_Token (Source : in Bounded_String;
                            Set    : in Maps.Character_Set;
                            Test   : in Membership;
                            First  : out Positive;
                            Last   : out Natural);
```

52
```
   -- String translation subprograms
```

53
```
      function Translate (Source  : in Bounded_String;
                          Mapping : in Maps.Character_Mapping)
         return Bounded_String;
```

54
```
      procedure Translate (Source  : in out Bounded_String;
                           Mapping : in Maps.Character_Mapping);
```

55
```
      function Translate (Source  : in Bounded_String;
                          Mapping : in Maps.Character_Mapping_Function)
         return Bounded_String;
```

56
```
      procedure Translate (Source  : in out Bounded_String;
                           Mapping : in Maps.Character_Mapping_Function);
```

57
```
   -- String transformation subprograms
```

58
```
      function Replace_Slice (Source   : in Bounded_String;
                              Low      : in Positive;
                              High     : in Natural;
                              By       : in String;
                              Drop     : in Truncation := Error)
         return Bounded_String;
```

59
```
      procedure Replace_Slice (Source   : in out Bounded_String;
                               Low      : in Positive;
                               High     : in Natural;
                               By       : in String;
                               Drop     : in Truncation := Error);
```

60
```
      function Insert (Source   : in Bounded_String;
                       Before   : in Positive;
                       New_Item : in String;
                       Drop     : in Truncation := Error)
         return Bounded_String;
```

61
```
   procedure Insert (Source   : in out Bounded_String;
                     Before   : in Positive;
                     New_Item : in String;
                     Drop     : in Truncation := Error);
```
62
```
   function Overwrite (Source    : in Bounded_String;
                       Position  : in Positive;
                       New_Item  : in String;
                       Drop      : in Truncation := Error)
      return Bounded_String;
```
63
```
   procedure Overwrite (Source    : in out Bounded_String;
                        Position  : in Positive;
                        New_Item  : in String;
                        Drop      : in Truncation := Error);
```
64
```
   function Delete (Source  : in Bounded_String;
                    From    : in Positive;
                    Through : in Natural)
      return Bounded_String;
```
65
```
   procedure Delete (Source  : in out Bounded_String;
                     From    : in Positive;
                     Through : in Natural);
```
66
```
--String selector subprograms
```
67
```
   function Trim (Source : in Bounded_String;
                  Side   : in Trim_End)
      return Bounded_String;
   procedure Trim (Source : in out Bounded_String;
                   Side   : in Trim_End);
```
68
```
   function Trim (Source : in Bounded_String;
                  Left   : in Maps.Character_Set;
                  Right  : in Maps.Character_Set)
      return Bounded_String;
```
69
```
   procedure Trim (Source : in out Bounded_String;
                   Left   : in Maps.Character_Set;
                   Right  : in Maps.Character_Set);
```
70
```
   function Head (Source : in Bounded_String;
                  Count  : in Natural;
                  Pad    : in Character  := Space;
                  Drop   : in Truncation := Error)
      return Bounded_String;
```
71
```
   procedure Head (Source : in out Bounded_String;
                   Count  : in Natural;
                   Pad    : in Character  := Space;
                   Drop   : in Truncation := Error);
```
72
```
   function Tail (Source : in Bounded_String;
                  Count  : in Natural;
                  Pad    : in Character  := Space;
                  Drop   : in Truncation := Error)
      return Bounded_String;
```
73
```
   procedure Tail (Source : in out Bounded_String;
                   Count  : in Natural;
                   Pad    : in Character  := Space;
                   Drop   : in Truncation := Error);
```
74
```
--String constructor subprograms
```
75
```
   function "*" (Left  : in Natural;
                 Right : in Character)
      return Bounded_String;
```
76
```
   function "*" (Left  : in Natural;
                 Right : in String)
      return Bounded_String;
```

77
```
      function "*" (Left  : in Natural;
                    Right : in Bounded_String)
         return Bounded_String;
```
78
```
      function Replicate (Count : in Natural;
                          Item  : in Character;
                          Drop  : in Truncation := Error)
         return Bounded_String;
```
79
```
      function Replicate (Count : in Natural;
                          Item  : in String;
                          Drop  : in Truncation := Error)
         return Bounded_String;
```
80
```
      function Replicate (Count : in Natural;
                          Item  : in Bounded_String;
                          Drop  : in Truncation := Error)
         return Bounded_String;
```
81
```
   private
       ... -- not specified by the language
   end Generic_Bounded_Length;
```
82
```
end Ada.Strings.Bounded;
```
82.a.1/2 *This paragraph was deleted.*{*8652/0097*} {*AI95-00115-01*} {*AI95-00344-01*}

83 Null_Bounded_String represents the null string. If an object of type Bounded_String is not otherwise initialized, it will be initialized to the same value as Null_Bounded_String.

84
```
function Length (Source : in Bounded_String) return Length_Range;
```
85 The Length function returns the length of the string represented by Source.

86
```
function To_Bounded_String (Source : in String;
                            Drop   : in Truncation := Error)
   return Bounded_String;
```
87 If Source'Length <= Max_Length then this function returns a Bounded_String that represents Source. Otherwise the effect depends on the value of Drop:

88 • If Drop=Left, then the result is a Bounded_String that represents the string comprising the rightmost Max_Length characters of Source.

89 • If Drop=Right, then the result is a Bounded_String that represents the string comprising the leftmost Max_Length characters of Source.

90 • If Drop=Error, then Strings.Length_Error is propagated.

91
```
function To_String (Source : in Bounded_String) return String;
```
92 To_String returns the String value with lower bound 1 represented by Source. If B is a Bounded_String, then B = To_Bounded_String(To_String(B)).

92.1/2
```
procedure Set_Bounded_String
   (Target :    out Bounded_String;
    Source : in     String;
    Drop   : in     Truncation := Error);
```
92.2/2 {*AI95-00301-01*} Equivalent to Target := To_Bounded_String (Source, Drop);

93 Each of the Append functions returns a Bounded_String obtained by concatenating the string or character given or represented by one of the parameters, with the string or character given or represented by the other parameter, and applying To_Bounded_String to the concatenation result string, with Drop as provided to the Append function.

94 Each of the procedures Append(Source, New_Item, Drop) has the same effect as the corresponding assignment Source := Append(Source, New_Item, Drop).

Each of the "&" functions has the same effect as the corresponding Append function, with Error as the Drop parameter. 95

96
```
function Element (Source : in Bounded_String;
                  Index  : in Positive)
   return Character;
```

Returns the character at position Index in the string represented by Source; propagates Index_Error if Index > Length(Source). 97

98
```
procedure Replace_Element (Source : in out Bounded_String;
                           Index  : in Positive;
                           By     : in Character);
```

Updates Source such that the character at position Index in the string represented by Source is By; propagates Index_Error if Index > Length(Source). 99

100
```
function Slice (Source : in Bounded_String;
                Low    : in Positive;
                High   : in Natural)
   return String;
```

{*8652/0049*} {*AI95-00128-01*} {*AI95-00238-01*} Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). The bounds of the returned string are Low and High.. 101/1

101.1/2
```
function Bounded_Slice
   (Source : in Bounded_String;
    Low    : in Positive;
    High   : in Natural)
       return Bounded_String;
```

{*AI95-00301-01*} Returns the slice at positions Low through High in the string represented by Source as a bounded string; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). 101.2/2

101.3/2
```
procedure Bounded_Slice
   (Source : in     Bounded_String;
    Target :    out Bounded_String;
    Low    : in     Positive;
    High   : in     Natural);
```

{*AI95-00301-01*} Equivalent to Target := Bounded_Slice (Source, Low, High); 101.4/2

Each of the functions "=", "<", ">", "<=", and ">=" returns the same result as the corresponding String operation applied to the String values given or represented by the two parameters. 102

Each of the search subprograms (Index, Index_Non_Blank, Count, Find_Token) has the same effect as the corresponding subprogram in Strings.Fixed applied to the string represented by the Bounded_String parameter. 103

Each of the Translate subprograms, when applied to a Bounded_String, has an analogous effect to the corresponding subprogram in Strings.Fixed. For the Translate function, the translation is applied to the string represented by the Bounded_String parameter, and the result is converted (via To_Bounded_String) to a Bounded_String. For the Translate procedure, the string represented by the Bounded_String parameter after the translation is given by the Translate function for fixed-length strings applied to the string represented by the original value of the parameter. 104

{*8652/0049*} {*AI95-00128-01*} Each of the transformation subprograms (Replace_Slice, Insert, Overwrite, Delete), selector subprograms (Trim, Head, Tail), and constructor functions ("*") has an effect based on its corresponding subprogram in Strings.Fixed, and Replicate is based on Fixed."*". In the case 105/1 of a function, the corresponding fixed-length string subprogram is applied to the string represented by the Bounded_String parameter. To_Bounded_String is applied the result string, with Drop (or Error in the case of Generic_Bounded_Length."*") determining the effect when the string length exceeds Max_Length. In the case of a procedure, the corresponding function in Strings.Bounded.Generic_-Bounded_Length is applied, with the result assigned into the Source parameter.

105.a/2 **Ramification:** {*AI95-00114-01*} The "/=" operations between Bounded_String and String, and between String and Bounded_String, are automatically defined based on the corresponding "=" operations.

Implementation Advice

106 Bounded string objects should not be implemented by implicit pointers and dynamic allocation.

106.a.1/2 **Implementation Advice:** Bounded string objects should not be implemented by implicit pointers and dynamic allocation.

106.a **Implementation Note:** The following is a possible implementation of the private part of the package:

106.b
```
type Bounded_String_Internals (Length : Length_Range := 0) is
   record
      Data : String(1..Length);
   end record;
```

106.c
```
type Bounded_String is
   record
      Data : Bounded_String_Internals;  -- Unconstrained
   end record;
```

106.d
```
Null_Bounded_String : constant Bounded_String :=
   (Data => (Length => 0,
             Data   => (1..0 => ' ')));
```

Inconsistencies With Ada 95

106.e/2 {*AI95-00238-01*} {*inconsistencies with Ada 95*} **Amendment Correction:** The bounds of the string returned from Slice are now defined. This is technically an inconsistency; if a program depended on some other lower bound for the string returned from Slice, it could fail when compiled with Ada 2005. Such code is not portable even between Ada 95 implementations, so it should be very rare.

Incompatibilities With Ada 95

106.f/2 {*AI95-00301-01*} {*incompatibilities with Ada 95*} Procedure Set_Bounded_String, two Bounded_Slice subprograms, and overloaded versions of Index and Index_Non_Blank are newly added to Strings.Bounded. If Strings.Bounded is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Strings.Bounded is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Wording Changes from Ada 95

106.g/2 {*8652/0049*} {*AI95-00128-01*} **Corrigendum:** Corrected the conditions for which Slice raises Index_Error.

106.h/2 {*8652/0049*} {*AI95-00128-01*} **Corrigendum:** Clarified the meaning of transformation, selector, and constructor subprograms by describing the effects of procedures and functions separately.

A.4.5 Unbounded-Length String Handling

1 The language-defined package Strings.Unbounded provides a private type Unbounded_String and a set of operations. An object of type Unbounded_String represents a String whose low bound is 1 and whose length can vary conceptually between 0 and Natural'Last. The subprograms for fixed-length string handling are either overloaded directly for Unbounded_String, or are modified as needed to reflect the flexibility in length. Since the Unbounded_String type is private, relevant constructor and selector operations are provided.

1.a **Reason:** The transformation operations for fixed- and bounded-length strings that are not necessarily length preserving are supplied for Unbounded_String as procedures as well as functions. This allows an implementation to do an initial allocation for an unbounded string and to avoid further allocations as long as the length does not exceed the allocated length.

Static Semantics

The library package Strings.Unbounded has the following declaration: 2

```
with Ada.Strings.Maps;                                                    3
package Ada.Strings.Unbounded is
   pragma Preelaborate(Unbounded);

{AI95-00161-01}    type Unbounded_String is private;                     4/2
   pragma Preelaborable_Initialization(Unbounded_String);

   Null_Unbounded_String : constant Unbounded_String;                     5

   function Length (Source : in Unbounded_String) return Natural;        6

   type String_Access is access all String;                               7
   procedure Free (X : in out String_Access);

-- Conversion, Concatenation, and Selection functions                     8

   function To_Unbounded_String (Source : in String)                      9
      return Unbounded_String;

   function To_Unbounded_String (Length : in Natural)                     10
      return Unbounded_String;

   function To_String (Source : in Unbounded_String) return String;      11

{AI95-00301-01}    procedure Set_Unbounded_String                        11.1/2
     (Target :    out Unbounded_String;
      Source : in     String);

   procedure Append (Source   : in out Unbounded_String;                  12
                     New_Item : in Unbounded_String);

   procedure Append (Source   : in out Unbounded_String;                  13
                     New_Item : in String);

   procedure Append (Source   : in out Unbounded_String;                  14
                     New_Item : in Character);

   function "&" (Left, Right : in Unbounded_String)                       15
      return Unbounded_String;

   function "&" (Left : in Unbounded_String; Right : in String)           16
      return Unbounded_String;

   function "&" (Left : in String; Right : in Unbounded_String)           17
      return Unbounded_String;

   function "&" (Left : in Unbounded_String; Right : in Character)        18
      return Unbounded_String;

   function "&" (Left : in Character; Right : in Unbounded_String)        19
      return Unbounded_String;

   function Element (Source : in Unbounded_String;                        20
                     Index  : in Positive)
      return Character;

   procedure Replace_Element (Source : in out Unbounded_String;           21
                              Index  : in Positive;
                              By     : in Character);

   function Slice (Source : in Unbounded_String;                          22
                   Low    : in Positive;
                   High   : in Natural)
      return String;

{AI95-00301-01}    function Unbounded_Slice                              22.1/2
      (Source : in Unbounded_String;
       Low    : in Positive;
       High   : in Natural)
         return Unbounded_String;
```

22.2/2
```
{AI95-00301-01}    procedure Unbounded_Slice
      (Source : in     Unbounded_String;
       Target :    out Unbounded_String;
       Low    : in     Positive;
       High   : in     Natural);
```
23
```
   function "="  (Left, Right : in Unbounded_String) return Boolean;
```
24
```
   function "="  (Left : in Unbounded_String; Right : in String)
     return Boolean;
```
25
```
   function "="  (Left : in String; Right : in Unbounded_String)
     return Boolean;
```
26
```
   function "<"  (Left, Right : in Unbounded_String) return Boolean;
```
27
```
   function "<"  (Left : in Unbounded_String; Right : in String)
     return Boolean;
```
28
```
   function "<"  (Left : in String; Right : in Unbounded_String)
     return Boolean;
```
29
```
   function "<=" (Left, Right : in Unbounded_String) return Boolean;
```
30
```
   function "<="  (Left : in Unbounded_String; Right : in String)
     return Boolean;
```
31
```
   function "<="  (Left : in String; Right : in Unbounded_String)
     return Boolean;
```
32
```
   function ">"  (Left, Right : in Unbounded_String) return Boolean;
```
33
```
   function ">"  (Left : in Unbounded_String; Right : in String)
     return Boolean;
```
34
```
   function ">"  (Left : in String; Right : in Unbounded_String)
     return Boolean;
```
35
```
   function ">=" (Left, Right : in Unbounded_String) return Boolean;
```
36
```
   function ">="  (Left : in Unbounded_String; Right : in String)
     return Boolean;
```
37
```
   function ">="  (Left : in String; Right : in Unbounded_String)
     return Boolean;
```
38 -- *Search subprograms*

38.1/2
```
{AI95-00301-01}    function Index (Source  : in Unbounded_String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping := Maps.Identity)
      return Natural;
```
38.2/2
```
{AI95-00301-01}    function Index (Source  : in Unbounded_String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping_Function)
      return Natural;
```
39
```
   function Index (Source   : in Unbounded_String;
                   Pattern  : in String;
                   Going    : in Direction := Forward;
                   Mapping  : in Maps.Character_Mapping
                                 := Maps.Identity)
      return Natural;
```
40
```
   function Index (Source   : in Unbounded_String;
                   Pattern  : in String;
                   Going    : in Direction := Forward;
                   Mapping  : in Maps.Character_Mapping_Function)
      return Natural;
```

40.1/2
```
{AI95-00301-01}    function Index (Source  : in Unbounded_String;
                   Set     : in Maps.Character_Set;
                   From    : in Positive;
                   Test    : in Membership := Inside;
                   Going    : in Direction := Forward)
      return Natural;
```

41
```
   function Index (Source : in Unbounded_String;
                   Set    : in Maps.Character_Set;
                   Test   : in Membership := Inside;
                   Going  : in Direction  := Forward) return Natural;
```

41.1/2
```
{AI95-00301-01}    function Index_Non_Blank (Source : in Unbounded_String;
                             From   : in Positive;
                             Going  : in Direction := Forward)
      return Natural;
```

42
```
   function Index_Non_Blank (Source : in Unbounded_String;
                             Going  : in Direction := Forward)
      return Natural;
```

43
```
   function Count (Source   : in Unbounded_String;
                   Pattern  : in String;
                   Mapping  : in Maps.Character_Mapping
                                := Maps.Identity)
      return Natural;
```

44
```
   function Count (Source   : in Unbounded_String;
                   Pattern  : in String;
                   Mapping  : in Maps.Character_Mapping_Function)
      return Natural;
```

45
```
   function Count (Source   : in Unbounded_String;
                   Set      : in Maps.Character_Set)
      return Natural;
```

46
```
   procedure Find_Token (Source : in Unbounded_String;
                         Set    : in Maps.Character_Set;
                         Test   : in Membership;
                         First  : out Positive;
                         Last   : out Natural);
```

47 -- *String translation subprograms*

48
```
   function Translate (Source  : in Unbounded_String;
                       Mapping : in Maps.Character_Mapping)
      return Unbounded_String;
```

49
```
   procedure Translate (Source  : in out Unbounded_String;
                        Mapping : in Maps.Character_Mapping);
```

50
```
   function Translate (Source  : in Unbounded_String;
                       Mapping : in Maps.Character_Mapping_Function)
      return Unbounded_String;
```

51
```
   procedure Translate (Source  : in out Unbounded_String;
                        Mapping : in Maps.Character_Mapping_Function);
```

52 -- *String transformation subprograms*

53
```
   function Replace_Slice (Source   : in Unbounded_String;
                           Low      : in Positive;
                           High     : in Natural;
                           By       : in String)
      return Unbounded_String;
```

54
```
   procedure Replace_Slice (Source   : in out Unbounded_String;
                            Low      : in Positive;
                            High     : in Natural;
                            By       : in String);
```

55
```
   function Insert (Source   : in Unbounded_String;
                    Before   : in Positive;
                    New_Item : in String)
      return Unbounded_String;
```

```
56        procedure Insert (Source   : in out Unbounded_String;
                            Before   : in Positive;
                            New_Item : in String);

57        function Overwrite (Source    : in Unbounded_String;
                              Position  : in Positive;
                              New_Item  : in String)
             return Unbounded_String;

58        procedure Overwrite (Source    : in out Unbounded_String;
                               Position  : in Positive;
                               New_Item  : in String);

59        function Delete (Source  : in Unbounded_String;
                           From    : in Positive;
                           Through : in Natural)
             return Unbounded_String;

60        procedure Delete (Source  : in out Unbounded_String;
                            From    : in Positive;
                            Through : in Natural);

61        function Trim (Source : in Unbounded_String;
                         Side   : in Trim_End)
             return Unbounded_String;

62        procedure Trim (Source : in out Unbounded_String;
                          Side   : in Trim_End);

63        function Trim (Source : in Unbounded_String;
                         Left   : in Maps.Character_Set;
                         Right  : in Maps.Character_Set)
             return Unbounded_String;

64        procedure Trim (Source : in out Unbounded_String;
                          Left   : in Maps.Character_Set;
                          Right  : in Maps.Character_Set);

65        function Head (Source : in Unbounded_String;
                         Count  : in Natural;
                         Pad    : in Character := Space)
             return Unbounded_String;

66        procedure Head (Source : in out Unbounded_String;
                          Count  : in Natural;
                          Pad    : in Character := Space);

67        function Tail (Source : in Unbounded_String;
                         Count  : in Natural;
                         Pad    : in Character := Space)
             return Unbounded_String;

68        procedure Tail (Source : in out Unbounded_String;
                          Count  : in Natural;
                          Pad    : in Character := Space);

69        function "*" (Left  : in Natural;
                        Right : in Character)
             return Unbounded_String;

70        function "*" (Left  : in Natural;
                        Right : in String)
             return Unbounded_String;

71        function "*" (Left  : in Natural;
                        Right : in Unbounded_String)
             return Unbounded_String;

72     private
          ... -- not specified by the language
       end Ada.Strings.Unbounded;
```

72.1/2 {*AI95-00360-01*} The type Unbounded_String needs finalization (see 7.6).

Null_Unbounded_String represents the null String. If an object of type Unbounded_String is not otherwise initialized, it will be initialized to the same value as Null_Unbounded_String. 73

The function Length returns the length of the String represented by Source. 74

The type String_Access provides a (non-private) access type for explicit processing of unbounded-length strings. The procedure Free performs an unchecked deallocation of an object of type String_Access. 75

The function To_Unbounded_String(Source : in String) returns an Unbounded_String that represents Source. The function To_Unbounded_String(Length : in Natural) returns an Unbounded_String that represents an uninitialized String whose length is Length. 76

The function To_String returns the String with lower bound 1 represented by Source. To_String and To_Unbounded_String are related as follows: 77

- If S is a String, then To_String(To_Unbounded_String(S)) = S. 78
- If U is an Unbounded_String, then To_Unbounded_String(To_String(U)) = U. 79

{*AI95-00301-01*} The procedure Set_Unbounded_String sets Target to an Unbounded_String that represents Source. 79.1/2

For each of the Append procedures, the resulting string represented by the Source parameter is given by the concatenation of the original value of Source and the value of New_Item. 80

Each of the "&" functions returns an Unbounded_String obtained by concatenating the string or character given or represented by one of the parameters, with the string or character given or represented by the other parameter, and applying To_Unbounded_String to the concatenation result string. 81

The Element, Replace_Element, and Slice subprograms have the same effect as the corresponding bounded-length string subprograms. 82

{*AI95-00301-01*} The function Unbounded_Slice returns the slice at positions Low through High in the string represented by Source as an Unbounded_String. The procedure Unbounded_Slice sets Target to the Unbounded_String representing the slice at positions Low through High in the string represented by Source. Both routines propagate Index_Error if Low > Length(Source)+1 or High > Length(Source). 82.1/2

Each of the functions "=", "<", ">", "<=", and ">=" returns the same result as the corresponding String operation applied to the String values given or represented by Left and Right. 83

Each of the search subprograms (Index, Index_Non_Blank, Count, Find_Token) has the same effect as the corresponding subprogram in Strings.Fixed applied to the string represented by the Unbounded_String parameter. 84

The Translate function has an analogous effect to the corresponding subprogram in Strings.Fixed. The translation is applied to the string represented by the Unbounded_String parameter, and the result is converted (via To_Unbounded_String) to an Unbounded_String. 85

Each of the transformation functions (Replace_Slice, Insert, Overwrite, Delete), selector functions (Trim, Head, Tail), and constructor functions ("*") is likewise analogous to its corresponding subprogram in Strings.Fixed. For each of the subprograms, the corresponding fixed-length string subprogram is applied to the string represented by the Unbounded_String parameter, and To_Unbounded_String is applied the result string. 86

For each of the procedures Translate, Replace_Slice, Insert, Overwrite, Delete, Trim, Head, and Tail, the resulting string represented by the Source parameter is given by the corresponding function for fixed-length strings applied to the string represented by Source's original value. 87

Implementation Requirements

88 No storage associated with an Unbounded_String object shall be lost upon assignment or scope exit.

88.a/2 **Implementation Note:** {*AI95-00301-01*} A sample implementation of the private part of the package and several of the subprograms appears in the Ada 95 Rationale.

Incompatibilities With Ada 95

88.b/2 {*AI95-00360-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** Type Unbounded_String is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and Unbounded_String does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as most Ada compilers have a controlled part in Unbounded_String, and thus would be illegal.

88.c/2 {*AI95-00301-01*} Procedure Set_Unbounded_String, two Unbounded_Slice subprograms, and overloaded versions of Index and Index_Non_Blank are newly added to Strings.Unbounded. If Strings.Unbounded is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Strings.Unbounded is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 95

88.d/2 {*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Added a pragma Preelaborable_Initialization to type Unbounded_String, so that it can be used to declare default-initialized objects in preelaborated units.

A.4.6 String-Handling Sets and Mappings

1 The language-defined package Strings.Maps.Constants declares Character_Set and Character_Mapping constants corresponding to classification and conversion functions in package Characters.Handling.

1.a **Discussion:** The Constants package is a child of Strings.Maps since it needs visibility of the private part of Strings.Maps in order to initialize the constants in a preelaborable way (i.e. via aggregates versus function calls).

Static Semantics

2 The library package Strings.Maps.Constants has the following declaration:

3/2
```
{AI95-00362-01} package Ada.Strings.Maps.Constants is
   pragma Pure(Constants);
```

4
```
   Control_Set           : constant Character_Set;
   Graphic_Set           : constant Character_Set;
   Letter_Set            : constant Character_Set;
   Lower_Set             : constant Character_Set;
   Upper_Set             : constant Character_Set;
   Basic_Set             : constant Character_Set;
   Decimal_Digit_Set     : constant Character_Set;
   Hexadecimal_Digit_Set : constant Character_Set;
   Alphanumeric_Set      : constant Character_Set;
   Special_Set           : constant Character_Set;
   ISO_646_Set           : constant Character_Set;
```

5
```
   Lower_Case_Map        : constant Character_Mapping;
     --Maps to lower case for letters, else identity
   Upper_Case_Map        : constant Character_Mapping;
     --Maps to upper case for letters, else identity
   Basic_Map             : constant Character_Mapping;
     --Maps to basic letter for letters, else identity
```

6
```
private
   ... -- not specified by the language
end Ada.Strings.Maps.Constants;
```

7 Each of these constants represents a correspondingly named set of characters or character mapping in Characters.Handling (see A.3.2).

Extensions to Ada 95

{*AI95-00362-01*} {*extensions to Ada 95*} Strings.Maps.Constants is now Pure, so it can be used in pure units. 7.a/2

A.4.7 Wide_String Handling

1/2 {*AI95-00302-03*} Facilities for handling strings of Wide_Character elements are found in the packages Strings.Wide_Maps, Strings.Wide_Fixed, Strings.Wide_Bounded, Strings.Wide_Unbounded, and Strings.Wide_Maps.Wide_Constants, and in the functions Strings.Wide_Hash, Strings.Wide_Fixed.-Wide_Hash, Strings.Wide_Bounded.Wide_Hash, and Strings.Wide_Unbounded.Wide_Hash. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements.

Static Semantics

2 The package Strings.Wide_Maps has the following declaration.

3
```
package Ada.Strings.Wide_Maps is
   pragma Preelaborate(Wide_Maps);
```

4/2
```
{AI95-00161-01}     -- Representation for a set of Wide_Character values:
   type Wide_Character_Set is private;
   pragma Preelaborable_Initialization(Wide_Character_Set);
```

5
```
   Null_Set : constant Wide_Character_Set;
```

6
```
   type Wide_Character_Range is
     record
         Low  : Wide_Character;
         High : Wide_Character;
     end record;
   -- Represents Wide_Character range Low..High
```

7
```
   type Wide_Character_Ranges is array (Positive range <>)
      of Wide_Character_Range;
```

8
```
   function To_Set    (Ranges : in Wide_Character_Ranges)
      return Wide_Character_Set;
```

9
```
   function To_Set    (Span   : in Wide_Character_Range)
      return Wide_Character_Set;
```

10
```
   function To_Ranges (Set    : in Wide_Character_Set)
      return Wide_Character_Ranges;
```

11
```
   function "="   (Left, Right : in Wide_Character_Set) return Boolean;
```

12
```
   function "not" (Right : in Wide_Character_Set)
      return Wide_Character_Set;
   function "and" (Left, Right : in Wide_Character_Set)
      return Wide_Character_Set;
   function "or"  (Left, Right : in Wide_Character_Set)
      return Wide_Character_Set;
   function "xor" (Left, Right : in Wide_Character_Set)
      return Wide_Character_Set;
   function "-"   (Left, Right : in Wide_Character_Set)
      return Wide_Character_Set;
```

13
```
   function Is_In (Element : in Wide_Character;
                   Set     : in Wide_Character_Set)
      return Boolean;
```

14
```
   function Is_Subset (Elements : in Wide_Character_Set;
                       Set      : in Wide_Character_Set)
      return Boolean;
```

15
```
   function "<=" (Left  : in Wide_Character_Set;
                  Right : in Wide_Character_Set)
      return Boolean renames Is_Subset;
```

```
16        -- Alternative representation for a set of Wide_Character values:
          subtype Wide_Character_Sequence is Wide_String;

17        function To_Set (Sequence  : in Wide_Character_Sequence)
             return Wide_Character_Set;

18        function To_Set (Singleton : in Wide_Character)
             return Wide_Character_Set;

19        function To_Sequence (Set  : in Wide_Character_Set)
             return Wide_Character_Sequence;

20/2   {AI95-00161-01}     -- Representation for a Wide_Character to Wide_Character mapping:
          type Wide_Character_Mapping is private;
          pragma Preelaborable_Initialization(Wide_Character_Mapping);

21        function Value (Map     : in Wide_Character_Mapping;
                          Element : in Wide_Character)
             return Wide_Character;

22        Identity : constant Wide_Character_Mapping;

23        function To_Mapping (From, To : in Wide_Character_Sequence)
             return Wide_Character_Mapping;

24        function To_Domain (Map : in Wide_Character_Mapping)
             return Wide_Character_Sequence;

25        function To_Range  (Map : in Wide_Character_Mapping)
             return Wide_Character_Sequence;

26        type Wide_Character_Mapping_Function is
             access function (From : in Wide_Character) return Wide_Character;

27     private
          ... -- not specified by the language
       end Ada.Strings.Wide_Maps;
```

28 The context clause for each of the packages Strings.Wide_Fixed, Strings.Wide_Bounded, and Strings.Wide_Unbounded identifies Strings.Wide_Maps instead of Strings.Maps.

29/2 {*AI95-00302-03*} For each of the packages Strings.Fixed, Strings.Bounded, Strings.Unbounded, and Strings.Maps.Constants, and for functions Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, and Strings.Unbounded.Hash, the corresponding wide string package has the same contents except that

- 30 Wide_Space replaces Space
- 31 Wide_Character replaces Character
- 32 Wide_String replaces String
- 33 Wide_Character_Set replaces Character_Set
- 34 Wide_Character_Mapping replaces Character_Mapping
- 35 Wide_Character_Mapping_Function replaces Character_Mapping_Function
- 36 Wide_Maps replaces Maps
- 37 Bounded_Wide_String replaces Bounded_String
- 38 Null_Bounded_Wide_String replaces Null_Bounded_String
- 39 To_Bounded_Wide_String replaces To_Bounded_String
- 40 To_Wide_String replaces To_String
- 40.1/2 {*AI95-00301-01*} Set_Bounded_Wide_String replaces Set_Bounded_String
- 41 Unbounded_Wide_String replaces Unbounded_String
- 42 Null_Unbounded_Wide_String replaces Null_Unbounded_String
- 43 Wide_String_Access replaces String_Access

- To_Unbounded_Wide_String replaces To_Unbounded_String 44
- {*AI95-00301-01*} Set_Unbounded_Wide_String replaces Set_Unbounded_String 44.1/2

The following additional declaration is present in Strings.Wide_Maps.Wide_Constants: 45

```
{AI95-00285-01} {AI95-00395-01} Character_Set : constant                46/2
Wide_Maps.Wide_Character_Set;
--Contains each Wide_Character value WC such that
--Characters.Conversions.Is_Character(WC) is True
```

{*AI95-00395-01*} Each Wide_Character_Set constant in the package Strings.Wide_Maps.Wide_Constants contains no values outside the Character portion of Wide_Character. Similarly, each Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Character. 46.1/2

{*AI95-00362-01*} Pragma Pure is replaced by pragma Preelaborate in Strings.Wide_Maps.Wide_Constants. 46.2/2

NOTES

12 {*Constraint_Error (raised by failure of run-time check)*} If a null Wide_Character_Mapping_Function is passed to any of the Wide_String handling subprograms, Constraint_Error is propagated. 47

This paragraph was deleted.{*AI95-00395-01*} 48/2

Incompatibilities With Ada 95

{*AI95-00301-01*} {*incompatibilities with Ada 95*} Various new operations are added to Strings.Wide_Fixed, Strings.Wide_Bounded, and Strings.Wide_Unbounded. If one of these packages is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 48.a/2

Extensions to Ada 95

{*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Added pragma Preelaborable_Initialization to types Wide_Character_Set and Wide_Character_Mapping, so that they can be used to declare default-initialized objects in preelaborated units. 48.b/2

Wording Changes from Ada 95

{*AI95-00285-01*} Corrected the description of Character_Set. 48.c/2

{*AI95-00302-03*} Added wide versions of Strings.Hash and Strings.Unbounded.Hash. 48.d/2

{*AI95-00362-01*} Added wording so that Strings.Wide_Maps.Wide_Constants does not change to Pure. 48.e/2

{*AI95-00395-01*} The second Note is now normative text, since there is no way to derive it from the other rules. It's a little weird given the use of Unicode character classifications in Ada 2005; but changing it would be inconsistent with Ada 95 and a one-to-one mapping isn't necessarily correct anyway. 48.f/2

A.4.8 Wide_Wide_String Handling

{*AI95-00285-01*} {*AI95-00395-01*} Facilities for handling strings of Wide_Wide_Character elements are found in the packages Strings.Wide_Wide_Maps, Strings.Wide_Wide_Fixed, Strings.Wide_Wide_-Bounded, Strings.Wide_Wide_Unbounded, and Strings.Wide_Wide_Maps.Wide_Wide_Constants, and in the functions Strings.Wide_Wide_Hash, Strings.Wide_Wide_Fixed.Wide_Wide_Hash, Strings.-Wide_Wide_Bounded.Wide_Wide_Hash, and Strings.Wide_Wide_Unbounded.Wide_Wide_Hash. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements. 1/2

Static Semantics

{*AI95-00285-01*} The library package Strings.Wide_Wide_Maps has the following declaration. 2/2

3/2
```
package Ada.Strings.Wide_Wide_Maps is
   pragma Preelaborate(Wide_Wide_Maps);
```
4/2
```
   -- Representation for a set of Wide_Wide_Character values:
   type Wide_Wide_Character_Set is private;
   pragma Preelaborable_Initialization(Wide_Wide_Character_Set);
```
5/2
```
   Null_Set : constant Wide_Wide_Character_Set;
```
6/2
```
   type Wide_Wide_Character_Range is
      record
         Low  : Wide_Wide_Character;
         High : Wide_Wide_Character;
      end record;
   -- Represents Wide_Wide_Character range Low..High
```
7/2
```
   type Wide_Wide_Character_Ranges is array (Positive range <>)
         of Wide_Wide_Character_Range;
```
8/2
```
   function To_Set (Ranges : in Wide_Wide_Character_Ranges)
         return Wide_Wide_Character_Set;
```
9/2
```
   function To_Set (Span : in Wide_Wide_Character_Range)
         return Wide_Wide_Character_Set;
```
10/2
```
   function To_Ranges (Set : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Ranges;
```
11/2
```
   function "=" (Left, Right : in Wide_Wide_Character_Set) return Boolean;
```
12/2
```
   function "not" (Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;
   function "and" (Left, Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;
   function "or" (Left, Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;
   function "xor" (Left, Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;
   function "-" (Left, Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;
```
13/2
```
   function Is_In (Element : in Wide_Wide_Character;
                   Set     : in Wide_Wide_Character_Set)
         return Boolean;
```
14/2
```
   function Is_Subset (Elements : in Wide_Wide_Character_Set;
                       Set      : in Wide_Wide_Character_Set)
         return Boolean;
```
15/2
```
   function "<=" (Left  : in Wide_Wide_Character_Set;
                  Right : in Wide_Wide_Character_Set)
         return Boolean renames Is_Subset;
```
16/2
```
   -- Alternative representation for a set of Wide_Wide_Character values:
   subtype Wide_Wide_Character_Sequence is Wide_Wide_String;
```
17/2
```
   function To_Set (Sequence : in Wide_Wide_Character_Sequence)
         return Wide_Wide_Character_Set;
```
18/2
```
   function To_Set (Singleton : in Wide_Wide_Character)
         return Wide_Wide_Character_Set;
```
19/2
```
   function To_Sequence (Set : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Sequence;
```
20/2
```
   -- Representation for a Wide_Wide_Character to Wide_Wide_Character
   -- mapping:
   type Wide_Wide_Character_Mapping is private;
   pragma Preelaborable_Initialization(Wide_Wide_Character_Mapping);
```
21/2
```
   function Value (Map     : in Wide_Wide_Character_Mapping;
                   Element : in Wide_Wide_Character)
         return Wide_Wide_Character;
```
22/2
```
   Identity : constant Wide_Wide_Character_Mapping;
```

```
      function To_Mapping (From, To : in Wide_Wide_Character_Sequence)        23/2
            return Wide_Wide_Character_Mapping;

      function To_Domain (Map : in Wide_Wide_Character_Mapping)                24/2
            return Wide_Wide_Character_Sequence;

      function To_Range (Map : in Wide_Wide_Character_Mapping)                 25/2
            return Wide_Wide_Character_Sequence;

      type Wide_Wide_Character_Mapping_Function is                             26/2
            access function (From : in Wide_Wide_Character)
            return Wide_Wide_Character;

   private                                                                     27/2
      ... -- not specified by the language
   end Ada.Strings.Wide_Wide_Maps;
```

{*AI95-00285-01*} The context clause for each of the packages Strings.Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, and Strings.Wide_Wide_Unbounded identifies Strings.Wide_Wide_Maps instead of Strings.Maps. 28/2

{*AI95-00285-01*} For each of the packages Strings.Fixed, Strings.Bounded, Strings.Unbounded, and Strings.Maps.Constants, and for functions Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, and Strings.Unbounded.Hash, the corresponding wide wide string package or function has the same contents except that 29/2

- Wide_Wide_Space replaces Space 30/2
- Wide_Wide_Character replaces Character 31/2
- Wide_Wide_String replaces String 32/2
- Wide_Wide_Character_Set replaces Character_Set 33/2
- Wide_Wide_Character_Mapping replaces Character_Mapping 34/2
- Wide_Wide_Character_Mapping_Function replaces Character_Mapping_Function 35/2
- Wide_Wide_Maps replaces Maps 36/2
- Bounded_Wide_Wide_String replaces Bounded_String 37/2
- Null_Bounded_Wide_Wide_String replaces Null_Bounded_String 38/2
- To_Bounded_Wide_Wide_String replaces To_Bounded_String 39/2
- To_Wide_Wide_String replaces To_String 40/2
- {*AI95-00301-01*} Set_Bounded_Wide_Wide_String replaces Set_Bounded_String 41/2
- Unbounded_Wide_Wide_String replaces Unbounded_String 42/2
- Null_Unbounded_Wide_Wide_String replaces Null_Unbounded_String 43/2
- Wide_Wide_String_Access replaces String_Access 44/2
- To_Unbounded_Wide_Wide_String replaces To_Unbounded_String 45/2
- {*AI95-00301-01*} Set_Unbounded_Wide_Wide_String replaces Set_Unbounded_String 46/2

{*AI95-00285-01*} {*AI95-00395-01*} The following additional declarations are present in Strings.Wide_Wide_Maps.Wide_Wide_Constants: 47/2

```
   Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;            48/2
   -- Contains each Wide_Wide_Character value WWC such that
   -- Characters.Conversions.Is_Character(WWC) is True
   Wide_Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
   -- Contains each Wide_Wide_Character value WWC such that
   -- Characters.Conversions.Is_Wide_Character(WWC) is True
```

49/2 {*AI95-00395-01*} Each Wide_Wide_Character_Set constant in the package Strings.Wide_Wide_Maps.-Wide_Wide_Constants contains no values outside the Character portion of Wide_Wide_Character. Similarly, each Wide_Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Wide_Character.

50/2 {*AI95-00395-01*} Pragma Pure is replaced by pragma Preelaborate in Strings.Wide_Wide_Maps.Wide_Wide_Constants.

NOTES

51/2 13 {*AI95-00285-01*} {*Constraint_Error (raised by failure of run-time check)*} If a null Wide_Wide_Character_Mapping_Function is passed to any of the Wide_Wide_String handling subprograms, Constraint_Error is propagated.

Extensions to Ada 95

51.a/2 {*AI95-00285-01*} {*AI95-00395-01*} {*extensions to Ada 95*} The double-wide string-handling packages (Strings.Wide_Wide_Maps, Strings.Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, Strings.Wide_Wide_Unbounded, and Strings.Wide_Wide_Maps.Wide_Wide_Constants), and functions Strings.Wide_Wide_Hash and Strings.Wide_Wide_Unbounded.Wide_Wide_Hash are new.

A.4.9 String Hashing

Static Semantics

1/2 {*AI95-00302-03*} The library function Strings.Hash has the following declaration:

2/2
```
with Ada.Containers;
function Ada.Strings.Hash (Key : String) return Containers.Hash_Type;
pragma Pure(Hash);
```

3/2 Returns an implementation-defined value which is a function of the value of Key. If *A* and *B* are strings such that *A* equals *B*, Hash(*A*) equals Hash(*B*).

3.a/2 **Implementation defined:** The values returned by Strings.Hash.

4/2 {*AI95-00302-03*} The library function Strings.Fixed.Hash has the following declaration:

5/2
```
with Ada.Containers, Ada.Strings.Hash;
function Ada.Strings.Fixed.Hash (Key : String) return Containers.Hash_Type
   renames Ada.Strings.Hash;
pragma Pure(Hash);
```

6/2 {*AI95-00302-03*} The generic library function Strings.Bounded.Hash has the following declaration:

7/2
```
with Ada.Containers;
generic
   with package Bounded is
                     new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Hash (Key : Bounded.Bounded_String)
   return Containers.Hash_Type;
pragma Preelaborate(Hash);
```

8/2 Strings.Bounded.Hash is equivalent to the function call Strings.Hash (Bounded.To_String (Key));

9/2 {*AI95-00302-03*} The library function Strings.Unbounded.Hash has the following declaration:

10/2
```
with Ada.Containers;
function Ada.Strings.Unbounded.Hash (Key : Unbounded_String)
   return Containers.Hash_Type;
pragma Preelaborate(Hash);
```

11/2 Strings.Unbounded.Hash is equivalent to the function call Strings.Hash (To_String (Key));

Implementation Advice

{*AI95-00302-03*} The Hash functions should be good hash functions, returning a wide spread of values for different string values. It should be unlikely for similar strings to return the same value. 12/2

Implementation Advice: Strings.Hash should be good a hash function, returning a wide spread of values for different string values, and similar strings should rarely return the same value. 12.a/2

Ramification: The other functions are defined in terms of Strings.Hash, so they don't need separate advice in the Annex. 12.b/2

Extensions to Ada 95

{*AI95-00302-03*} {*extensions to Ada 95*} The Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, and Strings.Unbounded.Hash functions are new. 12.c/2

A.5 The Numerics Packages

The library package Numerics is the parent of several child units that provide facilities for mathematical computation. One child, the generic package Generic_Elementary_Functions, is defined in A.5.1, together with nongeneric equivalents; two others, the package Float_Random and the generic package Discrete_Random, are defined in A.5.2. Additional (optional) children are defined in Annex G, "Numerics". 1

Static Semantics

This paragraph was deleted. 2/1

3/2
```
{AI95-00388-01} package Ada.Numerics is
   pragma Pure(Numerics);
   Argument_Error : exception;
   Pi : constant :=
          3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
   π  : constant := Pi;
   e  : constant :=
          2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;
```

The Argument_Error exception is raised by a subprogram in a child unit of Numerics to signal that one or more of the actual subprogram parameters are outside the domain of the corresponding mathematical function. 4

Implementation Permissions

The implementation may specify the values of Pi and e to a larger number of significant digits. 5

Reason: 51 digits seem more than adequate for all present computers; converted to binary, the values given above are accurate to more than 160 bits. Nevertheless, the permission allows implementations to accommodate unforeseen hardware advances. 5.a

Extensions to Ada 83

{*extensions to Ada 83*} Numerics and its children were not predefined in Ada 83. 5.b

Extensions to Ada 95

{*AI95-00388-01*} {*extensions to Ada 95*} The alternative declaration of π is new. 5.c/2

A.5.1 Elementary Functions

Implementation-defined approximations to the mathematical functions known as the "elementary functions" are provided by the subprograms in Numerics.Generic_Elementary_Functions. Nongeneric 1 equivalents of this generic package for each of the predefined floating point types are also provided as children of Numerics.

1.a **Implementation defined:** The accuracy actually achieved by the elementary functions.

Static Semantics

2 The generic library package Numerics.Generic_Elementary_Functions has the following declaration:

3
```
generic
   type Float_Type is digits <>;

package Ada.Numerics.Generic_Elementary_Functions is
   pragma Pure(Generic_Elementary_Functions);
```

4
```
   function Sqrt    (X           : Float_Type'Base) return Float_Type'Base;
   function Log     (X           : Float_Type'Base) return Float_Type'Base;
   function Log     (X, Base     : Float_Type'Base) return Float_Type'Base;
   function Exp     (X           : Float_Type'Base) return Float_Type'Base;
   function "**"    (Left, Right : Float_Type'Base) return Float_Type'Base;
```

5
```
   function Sin     (X           : Float_Type'Base) return Float_Type'Base;
   function Sin     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Cos     (X           : Float_Type'Base) return Float_Type'Base;
   function Cos     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Tan     (X           : Float_Type'Base) return Float_Type'Base;
   function Tan     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Cot     (X           : Float_Type'Base) return Float_Type'Base;
   function Cot     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
```

6
```
   function Arcsin  (X           : Float_Type'Base) return Float_Type'Base;
   function Arcsin  (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Arccos  (X           : Float_Type'Base) return Float_Type'Base;
   function Arccos  (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Arctan  (Y           : Float_Type'Base;
                     X           : Float_Type'Base := 1.0)
                                                    return Float_Type'Base;
   function Arctan  (Y           : Float_Type'Base;
                     X           : Float_Type'Base := 1.0;
                     Cycle       : Float_Type'Base) return Float_Type'Base;
   function Arccot  (X           : Float_Type'Base;
                     Y           : Float_Type'Base := 1.0)
                                                    return Float_Type'Base;
   function Arccot  (X           : Float_Type'Base;
                     Y           : Float_Type'Base := 1.0;
                     Cycle       : Float_Type'Base) return Float_Type'Base;
```

7
```
   function Sinh    (X           : Float_Type'Base) return Float_Type'Base;
   function Cosh    (X           : Float_Type'Base) return Float_Type'Base;
   function Tanh    (X           : Float_Type'Base) return Float_Type'Base;
   function Coth    (X           : Float_Type'Base) return Float_Type'Base;
   function Arcsinh (X           : Float_Type'Base) return Float_Type'Base;
   function Arccosh (X           : Float_Type'Base) return Float_Type'Base;
   function Arctanh (X           : Float_Type'Base) return Float_Type'Base;
   function Arccoth (X           : Float_Type'Base) return Float_Type'Base;
```

8
```
end Ada.Numerics.Generic_Elementary_Functions;
```

9/1 {*8652/0020*} {*AI95-00126-01*} The library package Numerics.Elementary_Functions is declared pure and defines the same subprograms as Numerics.Generic_Elementary_Functions, except that the predefined type Float is systematically substituted for Float_Type'Base throughout. Nongeneric equivalents of Numerics.Generic_Elementary_Functions for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Elementary_Functions, Numerics.Long_Elementary_-Functions, etc.

9.a **Reason:** The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics.

10 The functions have their usual mathematical meanings. When the Base parameter is specified, the Log function computes the logarithm to the given base; otherwise, it computes the natural logarithm. When the Cycle parameter is specified, the parameter X of the forward trigonometric functions (Sin, Cos, Tan, and Cot) and the results of the inverse trigonometric functions (Arcsin, Arccos, Arctan, and Arccot) are measured in units such that a full cycle of revolution has the given value; otherwise, they are measured in radians.

11 The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

- 12 The results of the Sqrt and Arccosh functions and that of the exponentiation operator are nonnegative.
- 13 The result of the Arcsin function is in the quadrant containing the point (1.0, *x*), where *x* is the value of the parameter X. This quadrant is I or IV; thus, the range of the Arcsin function is approximately –π/2.0 to π/2.0 (–Cycle/4.0 to Cycle/4.0, if the parameter Cycle is specified).
- 14 The result of the Arccos function is in the quadrant containing the point (*x*, 1.0), where *x* is the value of the parameter X. This quadrant is I or II; thus, the Arccos function ranges from 0.0 to approximately π (Cycle/2.0, if the parameter Cycle is specified).
- 15 The results of the Arctan and Arccot functions are in the quadrant containing the point (*x*, *y*), where *x* and *y* are the values of the parameters X and Y, respectively. This may be any quadrant (I through IV) when the parameter X (resp., Y) of Arctan (resp., Arccot) is specified, but it is restricted to quadrants I and IV (resp., I and II) when that parameter is omitted. Thus, the range when that parameter is specified is approximately –π to π (–Cycle/2.0 to Cycle/2.0, if the parameter Cycle is specified); when omitted, the range of Arctan (resp., Arccot) is that of Arcsin (resp., Arccos), as given above. When the point (*x*, *y*) lies on the negative x-axis, the result approximates
  - 16 π (resp., –π) when the sign of the parameter Y is positive (resp., negative), if Float_Type'Signed_Zeros is True;
  - 17 π, if Float_Type'Signed_Zeros is False.

18 (In the case of the inverse trigonometric functions, in which a result lying on or near one of the axes may not be exactly representable, the approximation inherent in computing the result may place it in an adjacent quadrant, close to but on the wrong side of the axis.)

Dynamic Semantics

19 The exception Numerics.Argument_Error is raised, signaling a parameter value outside the domain of the corresponding mathematical function, in the following cases:

- 20 by any forward or inverse trigonometric function with specified cycle, when the value of the parameter Cycle is zero or negative;
- 21 by the Log function with specified base, when the value of the parameter Base is zero, one, or negative;
- 22 by the Sqrt and Log functions, when the value of the parameter X is negative;
- 23 by the exponentiation operator, when the value of the left operand is negative or when both operands have the value zero;
- 24 by the Arcsin, Arccos, and Arctanh functions, when the absolute value of the parameter X exceeds one;
- 25 by the Arctan and Arccot functions, when the parameters X and Y both have the value zero;
- 26 by the Arccosh function, when the value of the parameter X is less than one; and

27 • by the Arccoth function, when the absolute value of the parameter X is less than one.

28 {*Division_Check* [partial]} {*check, language-defined (Division_Check)*} {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised, signaling a pole of the mathematical function (analogous to dividing by zero), in the following cases, provided that Float_Type'Machine_Overflows is True:

29 • by the Log, Cot, and Coth functions, when the value of the parameter X is zero;

30 • by the exponentiation operator, when the value of the left operand is zero and the value of the exponent is negative;

31 • by the Tan function with specified cycle, when the value of the parameter X is an odd multiple of the quarter cycle;

32 • by the Cot function with specified cycle, when the value of the parameter X is zero or a multiple of the half cycle; and

33 • by the Arctanh and Arccoth functions, when the absolute value of the parameter X is one.

34 {*Constraint_Error (raised by failure of run-time check)*} [Constraint_Error can also be raised when a finite result overflows (see G.2.4); this may occur for parameter values sufficiently *near* poles, and, in the case of some of the functions, for parameter values with sufficiently large magnitudes.]{*unspecified* [partial]} When Float_Type'Machine_Overflows is False, the result at poles is unspecified.

34.a **Reason:** The purpose of raising Constraint_Error (rather than Numerics.Argument_Error) at the poles of a function, when Float_Type'Machine_Overflows is True, is to provide continuous behavior as the actual parameters of the function approach the pole and finally reach it.

34.b **Discussion:** It is anticipated that an Ada binding to IEC 559:1989 will be developed in the future. As part of such a binding, the Machine_Overflows attribute of a conformant floating point type will be specified to yield False, which will permit both the predefined arithmetic operations and implementations of the elementary functions to deliver signed infinities (and set the overflow flag defined by the binding) instead of raising Constraint_Error in overflow situations, when traps are disabled. Similarly, it is appropriate for the elementary functions to deliver signed infinities (and set the zero-divide flag defined by the binding) instead of raising Constraint_Error at poles, when traps are disabled. Finally, such a binding should also specify the behavior of the elementary functions, when sensible, given parameters with infinite values.

35 When one parameter of a function with multiple parameters represents a pole and another is outside the function's domain, the latter takes precedence (i.e., Numerics.Argument_Error is raised).

Implementation Requirements

36 In the implementation of Numerics.Generic_Elementary_Functions, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Float_Type.

36.a **Implementation Note:** Implementations of Numerics.Generic_Elementary_Functions written in Ada should therefore avoid declaring local variables of subtype Float_Type; the subtype Float_Type'Base should be used instead.

37 {*prescribed result (for the evaluation of an elementary function)*} In the following cases, evaluation of an elementary function shall yield the *prescribed result*, provided that the preceding rules do not call for an exception to be raised:

38 • When the parameter X has the value zero, the Sqrt, Sin, Arcsin, Tan, Sinh, Arcsinh, Tanh, and Arctanh functions yield a result of zero, and the Exp, Cos, and Cosh functions yield a result of one.

39 • When the parameter X has the value one, the Sqrt function yields a result of one, and the Log, Arccos, and Arccosh functions yield a result of zero.

40 • When the parameter Y has the value zero and the parameter X has a positive value, the Arctan and Arccot functions yield a result of zero.

- The results of the Sin, Cos, Tan, and Cot functions with specified cycle are exact when the mathematical result is zero; those of the first two are also exact when the mathematical result is ± 1.0. 41
- Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand. Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero. 42

Other accuracy requirements for the elementary functions, which apply only in implementations conforming to the Numerics Annex, and then only in the "strict" mode defined there (see G.2), are given in G.2.4. 43

When Float_Type'Signed_Zeros is True, the sign of a zero result shall be as follows: 44

- A prescribed zero result delivered *at the origin* by one of the odd functions (Sin, Arcsin, Sinh, Arcsinh, Tan, Arctan or Arccot as a function of Y when X is fixed and positive, Tanh, and Arctanh) has the sign of the parameter X (Y, in the case of Arctan or Arccot). 45
- A prescribed zero result delivered by one of the odd functions *away from the origin*, or by some other elementary function, has an implementation-defined sign. 46

  Implementation defined: The sign of a zero result from some of the operators or functions in Numerics.Generic_Elementary_Functions, when Float_Type'Signed_Zeros is True. 46.a

- [A zero result that is not a prescribed result (i.e., one that results from rounding or underflow) has the correct mathematical sign.] 47

  Reason: This is a consequence of the rules specified in IEC 559:1989 as they apply to underflow situations with traps disabled. 47.a

Implementation Permissions

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type. 48

Wording Changes from Ada 83

The semantics of Numerics.Generic_Elementary_Functions differs from Generic_Elementary_Functions as defined in ISO/IEC DIS 11430 (for Ada 83) in the following ways: 48.a

- The generic package is a child unit of the package defining the Argument_Error exception. 48.b
- DIS 11430 specified names for the nongeneric equivalents, if provided. Here, those nongeneric equivalents are required. 48.c
- Implementations are not allowed to impose an optional restriction that the generic actual parameter associated with Float_Type be unconstrained. (In view of the ability to declare variables of subtype Float_Type'Base in implementations of Numerics.Generic_Elementary_Functions, this flexibility is no longer needed.) 48.d
- The sign of a prescribed zero result at the origin of the odd functions is specified, when Float_Type'Signed_Zeros is True. This conforms with recommendations of Kahan and other numerical analysts. 48.e
- The dependence of Arctan and Arccot on the sign of a parameter value of zero is tied to the value of Float_Type'Signed_Zeros. 48.f
- Sqrt is prescribed to yield a result of one when its parameter has the value one. This guarantee makes it easier to achieve certain prescribed results of the complex elementary functions (see G.1.2, "Complex Elementary Functions"). 48.g
- Conformance to accuracy requirements is conditional. 48.h

Wording Changes from Ada 95

{*8652/0020*} {*AI95-00126-01*} **Corrigendum:** Explicitly stated that the nongeneric equivalents of Generic_Elementary_Functions are pure. 48.i/2

A.5.2 Random Number Generation

1 [Facilities for the generation of pseudo-random floating point numbers are provided in the package Numerics.Float_Random; the generic package Numerics.Discrete_Random provides similar facilities for the generation of pseudo-random integers and pseudo-random values of enumeration types. {*random number*} For brevity, pseudo-random values of any of these types are called *random numbers*.

2 Some of the facilities provided are basic to all applications of random numbers. These include a limited private type each of whose objects serves as the generator of a (possibly distinct) sequence of random numbers; a function to obtain the “next” random number from a given sequence of random numbers (that is, from its generator); and subprograms to initialize or reinitialize a given generator to a time-dependent state or a state denoted by a single integer.

3 Other facilities are provided specifically for advanced applications. These include subprograms to save and restore the state of a given generator; a private type whose objects can be used to hold the saved state of a generator; and subprograms to obtain a string representation of a given generator state, or, given such a string representation, the corresponding state.]

3.a **Discussion:** These facilities support a variety of requirements ranging from repeatable sequences (for debugging) to unique sequences in each execution of a program.

Static Semantics

4 The library package Numerics.Float_Random has the following declaration:

```
5    package Ada.Numerics.Float_Random is

6       -- Basic facilities

7       type Generator is limited private;

8       subtype Uniformly_Distributed is Float range 0.0 .. 1.0;
        function Random (Gen : Generator) return Uniformly_Distributed;

9       procedure Reset (Gen       : in Generator;
                         Initiator : in Integer);
        procedure Reset (Gen       : in Generator);

10      -- Advanced facilities

11      type State is private;

12      procedure Save  (Gen        : in  Generator;
                         To_State   : out State);
        procedure Reset (Gen        : in  Generator;
                         From_State : in  State);

13      Max_Image_Width : constant := implementation-defined integer value;

14      function Image (Of_State    : State)  return String;
        function Value (Coded_State : String) return State;

15   private
        ... -- not specified by the language
     end Ada.Numerics.Float_Random;
```

15.1/2 {*AI95-00360-01*} The type Generator needs finalization (see 7.6).

16 The generic library package Numerics.Discrete_Random has the following declaration:

```
17
     generic
        type Result_Subtype is (<>);
     package Ada.Numerics.Discrete_Random is

18      -- Basic facilities

19      type Generator is limited private;

20      function Random (Gen : Generator) return Result_Subtype;
```

21
```
      procedure Reset (Gen       : in Generator;
                       Initiator : in Integer);
      procedure Reset (Gen       : in Generator);
```

22
```
      -- Advanced facilities
```

23
```
      type State is private;
```

24
```
      procedure Save  (Gen        : in  Generator;
                       To_State   : out State);
      procedure Reset (Gen        : in  Generator;
                       From_State : in  State);
```

25
```
      Max_Image_Width : constant := implementation-defined integer value;
```

26
```
      function Image (Of_State    : State)  return String;
      function Value (Coded_State : String) return State;
```

27
```
   private
      ... -- not specified by the language
   end Ada.Numerics.Discrete_Random;
```

27.a **Implementation defined:** The value of Numerics.Float_Random.Max_Image_Width.

27.b **Implementation defined:** The value of Numerics.Discrete_Random.Max_Image_Width.

27.c/1 **Implementation Note:** {*8652/0097*} {*AI95-00115-01*} The following is a possible implementation of the private part of Numerics.Float_Random (assuming the presence of “**with** Ada.Finalization;” as a context clause):

27.d
```
type State is ...;
type Access_State is access State;
type Generator is new Finalization.Limited_Controlled with
   record
      S : Access_State := new State'(...);
   end record;
procedure Finalize (G : in out Generator);
```

27.d.1/2 {*8652/0097*} {*AI95-00115-01*} {*AI95-00344-01*} Numerics.Discrete_Random.Generator also can be implemented this way.

27.e Clearly some level of indirection is required in the implementation of a Generator, since the parameter mode is **in** for all operations on a Generator. For this reason, Numerics.Float_Random and Numerics.Discrete_Random cannot be declared pure.

27.1/2 {*AI95-00360-01*} The type Generator needs finalization (see 7.6) in every instantiation of Numerics.Discrete_Random.

28 An object of the limited private type Generator is associated with a sequence of random numbers. Each generator has a hidden (internal) state, which the operations on generators use to determine the position in the associated sequence. {*unspecified* [partial]} All generators are implicitly initialized to an unspecified state that does not vary from one program execution to another; they may also be explicitly initialized, or reinitialized, to a time-dependent state, to a previously saved state, or to a state uniquely denoted by an integer value.

28.a **Discussion:** The repeatability provided by the implicit initialization may be exploited for testing or debugging purposes.

29 An object of the private type State can be used to hold the internal state of a generator. Such objects are only needed if the application is designed to save and restore generator states or to examine or manufacture them.

30 The operations on generators affect the state and therefore the future values of the associated sequence. The semantics of the operations on generators and states are defined below.

31
```
   function Random (Gen : Generator) return Uniformly_Distributed;
   function Random (Gen : Generator) return Result_Subtype;
```

32 Obtains the “next” random number from the given generator, relative to its current state, according to an implementation-defined algorithm. The result of the function in Numerics.Float_Random is delivered as a value of the subtype Uniformly_Distributed, which is a

subtype of the predefined type Float having a range of 0.0 .. 1.0. The result of the function in an instantiation of Numerics.Discrete_Random is delivered as a value of the generic formal subtype Result_Subtype.

32.a/2 *This paragraph was deleted.*

32.a.1/2 **Discussion:** The algorithm is the subject of a Documentation Requirement, so we don't separately summarize this implementation-defined item.

32.b **Reason:** The requirement for a level of indirection in accessing the internal state of a generator arises from the desire to make Random a function, rather than a procedure.

33
```
procedure Reset (Gen       : in Generator;
                 Initiator : in Integer);
procedure Reset (Gen       : in Generator);
```

34 {*unspecified* [partial]} Sets the state of the specified generator to one that is an unspecified function of the value of the parameter Initiator (or to a time-dependent state, if only a generator parameter is specified). {*Time-dependent Reset procedure (of the random number generator)*} The latter form of the procedure is known as the *time-dependent Reset procedure*.

34.a **Implementation Note:** The time-dependent Reset procedure can be implemented by mapping the current time and date as determined by the system clock into a state, but other implementations are possible. For example, a white-noise generator or a radioactive source can be used to generate time-dependent states.

35
```
procedure Save  (Gen        : in  Generator;
                 To_State   : out State);
procedure Reset (Gen        : in  Generator;
                 From_State : in  State);
```

36 Save obtains the current state of a generator. Reset gives a generator the specified state. A generator that is reset to a state previously obtained by invoking Save is restored to the state it had when Save was invoked.

37
```
function Image (Of_State    : State)  return String;
function Value (Coded_State : String) return State;
```

38 Image provides a representation of a state coded (in an implementation-defined way) as a string whose length is bounded by the value of Max_Image_Width. Value is the inverse of Image: Value(Image(S)) = S for each state S that can be obtained from a generator by invoking Save.

38.a **Implementation defined:** The string representation of a random number generator's state.

Dynamic Semantics

39 {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} {*Constraint_Error (raised by failure of run-time check)*} Instantiation of Numerics.Discrete_Random with a subtype having a null range raises Constraint_Error.

40/1 *This paragraph was deleted.*{*8652/0050*} {*AI95-00089*}

Bounded (Run-Time) Errors

40.1/1 {*8652/0050*} {*AI95-00089*} It is a bounded error to invoke Value with a string that is not the image of any generator state. {*Program_Error (raised by failure of run-time check)*} {*Constraint_Error (raised by failure of run-time check)*} If the error is detected, Constraint_Error or Program_Error is raised. Otherwise, a call to Reset with the resulting state will produce a generator such that calls to Random with this generator will produce a sequence of values of the appropriate subtype, but which might not be random in character. That is, the sequence of values might not fulfill the implementation requirements of this subclause.

Implementation Requirements

41 A sufficiently long sequence of random numbers obtained by successive calls to Random is approximately uniformly distributed over the range of the result subtype.

42 The Random function in an instantiation of Numerics.Discrete_Random is guaranteed to yield each value in its result subtype in a finite number of calls, provided that the number of such values does not exceed 2^{15}.

43 Other performance requirements for the random number generator, which apply only in implementations conforming to the Numerics Annex, and then only in the "strict" mode defined there (see G.2), are given in G.2.5.

Documentation Requirements

44 No one algorithm for random number generation is best for all applications. To enable the user to determine the suitability of the random number generators for the intended application, the implementation shall describe the algorithm used and shall give its period, if known exactly, or a lower bound on the period, if the exact period is unknown. Periods that are so long that the periodicity is unobservable in practice can be described in such terms, without giving a numerical bound.

44.a/2 **Documentation Requirement:** The algorithm used for random number generation, including a description of its period.

45 The implementation also shall document the minimum time interval between calls to the time-dependent Reset procedure that are guaranteed to initiate different sequences, and it shall document the nature of the strings that Value will accept without raising Constraint_Error.

45.a/2 *This paragraph was deleted.*

45.b/2 **Documentation Requirement:** The minimum time interval between calls to the time-dependent Reset procedure that is guaranteed to initiate different random number sequences.

Implementation Advice

46 Any storage associated with an object of type Generator should be reclaimed on exit from the scope of the object.

46.a.1/2 **Implementation Advice:** Any storage associated with an object of type Generator of the random number packages should be reclaimed on exit from the scope of the object.

46.a **Ramification:** A level of indirection is implicit in the semantics of the operations, given that they all take parameters of mode **in**. This implies that the full type of Generator probably should be a controlled type, with appropriate finalization to reclaim any heap-allocated storage.

47 If the generator period is sufficiently long in relation to the number of distinct initiator values, then each possible value of Initiator passed to Reset should initiate a sequence of random numbers that does not, in a practical sense, overlap the sequence initiated by any other value. If this is not possible, then the mapping between initiator values and generator states should be a rapidly varying function of the initiator value.

47.a/2 **Implementation Advice:** Each value of Initiator passed to Reset for the random number packages should initiate a distinct sequence of random numbers, or, if that is not possible, be at least a rapidly varying function of the initiator value.

NOTES

48 14 If two or more tasks are to share the same generator, then the tasks have to synchronize their access to the generator as for any shared variable (see 9.10).

49 15 Within a given implementation, a repeatable random number sequence can be obtained by relying on the implicit initialization of generators or by explicitly initializing a generator with a repeatable initiator value. Different sequences of random numbers can be obtained from a given generator in different program executions by explicitly initializing the generator to a time-dependent state.

50 16 A given implementation of the Random function in Numerics.Float_Random may or may not be capable of delivering the values 0.0 or 1.0. Portable applications should assume that these values, or values sufficiently close to them to behave indistinguishably from them, can occur. If a sequence of random integers from some fixed range is needed, the application should use the Random function in an appropriate instantiation of Numerics.Discrete_Random, rather than transforming the result of the Random function in Numerics.Float_Random. However, some applications with unusual requirements, such as for a sequence of random integers each drawn from a different range, will find it more convenient to transform the result of the floating point Random function. For M ≥ 1, the expression

51
```
   Integer(Float(M) * Random(G)) mod M
```

52 transforms the result of Random(G) to an integer uniformly distributed over the range 0 .. M–1; it is valid even if Random delivers 0.0 or 1.0. Each value of the result range is possible, provided that M is not too large. Exponentially distributed (floating point) random numbers with mean and standard deviation 1.0 can be obtained by the transformation

53/2
```
{AI95-00434-01}    -Log(Random(G) + Float'Model_Small)
```

54 where Log comes from Numerics.Elementary_Functions (see A.5.1); in this expression, the addition of Float'Model_Small avoids the exception that would be raised were Log to be given the value zero, without affecting the result (in most implementations) when Random returns a nonzero value.

Examples

55 *Example of a program that plays a simulated dice game:*

56
```
with Ada.Numerics.Discrete_Random;
procedure Dice_Game is
   subtype Die is Integer range 1 .. 6;
   subtype Dice is Integer range 2*Die'First .. 2*Die'Last;
   package Random_Die is new Ada.Numerics.Discrete_Random (Die);
   use Random_Die;
   G : Generator;
   D : Dice;
begin
   Reset (G);  -- Start the generator in a unique state in each run
   loop
      -- Roll a pair of dice; sum and process the results
      D := Random(G) + Random(G);
      ...
   end loop;
end Dice_Game;
```

57 *Example of a program that simulates coin tosses:*

58
```
with Ada.Numerics.Discrete_Random;
procedure Flip_A_Coin is
   type Coin is (Heads, Tails);
   package Random_Coin is new Ada.Numerics.Discrete_Random (Coin);
   use Random_Coin;
   G : Generator;
begin
   Reset (G);  -- Start the generator in a unique state in each run
   loop
      -- Toss a coin and process the result
      case Random(G) is
          when Heads =>
             ...
          when Tails =>
             ...
      end case;
   ...
   end loop;
end Flip_A_Coin;
```

Example of a parallel simulation of a physical system, with a separate generator of event probabilities in each task: 59

60
```
with Ada.Numerics.Float_Random;
procedure Parallel_Simulation is
   use Ada.Numerics.Float_Random;
   task type Worker is
      entry Initialize_Generator (Initiator : in Integer);
      ...
   end Worker;
   W : array (1 .. 10) of Worker;
   task body Worker is
      G : Generator;
      Probability_Of_Event : Uniformly_Distributed;
   begin
      accept Initialize_Generator (Initiator : in Integer) do
         Reset (G, Initiator);
      end Initialize_Generator;
      loop
         ...
         Probability_Of_Event := Random(G);
         ...
      end loop;
   end Worker;
begin
   -- Initialize the generators in the Worker tasks to different states
   for I in W'Range loop
      W(I).Initialize_Generator (I);
   end loop;
   ... -- Wait for the Worker tasks to terminate
end Parallel_Simulation;
```

NOTES

17 *Notes on the last example:* Although each Worker task initializes its generator to a different state, those states will be the same in every execution of the program. The generator states can be initialized uniquely in each program execution by instantiating Ada.Numerics.Discrete_Random for the type Integer in the main procedure, resetting the generator obtained from that instance to a time-dependent state, and then using random integers obtained from that generator to initialize the generators in each Worker task. 61

Incompatibilities With Ada 95

{*AI95-00360-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** Type Generator in Numerics.Float_Random and in an instance of Numerics.Discrete_Random is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and Generator does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in Generator, and thus would be illegal. 61.a/2

Wording Changes from Ada 95

{*8652/0050*} {*AI95-00089-01*} **Corrigendum:** Made the passing of an incorrect Image of a generator a bounded error, as it may not be practical to check for problems (if a generator consists of several related values). 61.b/2

A.5.3 Attributes of Floating Point Types

Static Semantics

{*representation-oriented attributes (of a floating point subtype)*} The following *representation-oriented attributes* are defined for every subtype S of a floating point type *T*. 1

S'Machine_Radix 2

Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*.

{*canonical form*} The values of other representation-oriented attributes of a floating point subtype, and of the “primitive function” attributes of a floating point subtype described later, are defined in terms of a 3 particular representation of nonzero values called the *canonical form*. The canonical form (for the type *T*) is the form

$\pm\, mantissa \cdot T\text{'Machine\_Radix}^{exponent}$

where

4 • *mantissa* is a fraction in the number base *T*'Machine_Radix, the first digit of which is nonzero, and

5 • *exponent* is an integer.

6 S'Machine_Mantissa
Yields the largest value of *p* such that every value expressible in the canonical form (for the type *T*), having a *p*-digit *mantissa* and an *exponent* between *T*'Machine_Emin and *T*'Machine_Emax, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

6.a **Ramification:** Values of a type held in an extended register are, in general, not machine numbers of the type, since they cannot be expressed in the canonical form with a sufficiently short *mantissa*.

7 S'Machine_Emin
Yields the smallest (most negative) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

8 S'Machine_Emax
Yields the largest (most positive) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

8.a **Ramification:** Note that the above definitions do not determine unique values for the representation-oriented attributes of floating point types. The implementation may choose any set of values that collectively satisfies the definitions.

9 S'Denorm Yields the value True if every value expressible in the form

$\pm\, mantissa \cdot T\text{'Machine\_Radix}^{T\text{'Machine\_Emin}}$

where *mantissa* is a nonzero *T*'Machine_Mantissa-digit fraction in the number base *T*'Machine_Radix, the first digit of which is zero, is a machine number (see 3.5.7) of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

10 {*denormalized number*} The values described by the formula in the definition of S'Denorm are called *denormalized numbers*. {*normalized number*} A nonzero machine number that is not a denormalized number is a *normalized number*. {*represented in canonical form*} {*canonical-form representation*} A normalized number *x* of a given type *T* is said to be *represented in canonical form* when it is expressed in the canonical form (for the type *T*) with a *mantissa* having *T*'Machine_Mantissa digits; the resulting form is the *canonical-form representation* of *x*.

10.a **Discussion:** The intent is that S'Denorm be True when such denormalized numbers exist and are generated in the circumstances defined by IEC 559:1989, though the latter requirement is not formalized here.

11 S'Machine_Rounds
Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

11.a **Discussion:** It is difficult to be more precise about what it means to round the result of a predefined operation. If the implementation does not use extended registers, so that every arithmetic result is necessarily a machine number, then rounding seems to imply two things:

11.b • S'Model_Mantissa = S'Machine_Mantissa, so that operand preperturbation never occurs;

- when the exact mathematical result is not a machine number, the result of a predefined operation must be the nearer of the two adjacent machine numbers. 11.c

Technically, this attribute should yield False when extended registers are used, since a few computed results will cross over the half-way point as a result of double rounding, if and when a value held in an extended register has to be reduced in precision to that of the machine numbers. It does not seem desirable to preclude the use of extended registers when S'Machine_Rounds could otherwise be True. 11.d

S'Machine_Overflows 12
Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

S'Signed_Zeros 13
Yields the value True if the hardware representation for the type *T* has the capability of representing both positively and negatively signed zeros, these being generated and used by the predefined operations of the type *T* as specified in IEC 559:1989; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

{*normalized exponent*} For every value *x* of a floating point type *T*, the *normalized exponent* of *x* is defined as follows: 14

- the normalized exponent of zero is (by convention) zero; 15
- for nonzero *x*, the normalized exponent of *x* is the unique integer *k* such that *T*'Machine_Radix^{k-1} ≤ |*x*| < *T*'Machine_Radix^{k}. 16

Ramification: The normalized exponent of a normalized number *x* is the value of *exponent* in the canonical-form representation of *x*. 16.a

The normalized exponent of a denormalized number is less than the value of *T*'Machine_Emin. 16.b

{*primitive function*} The following *primitive function attributes* are defined for any subtype S of a floating point type *T*. 17

S'Exponent S'Exponent denotes a function with the following specification: 18

```
function S'Exponent (X : T)
  return universal_integer
```
19

The function yields the normalized exponent of *X*. 20

S'Fraction S'Fraction denotes a function with the following specification: 21

```
function S'Fraction (X : T)
  return T
```
22

The function yields the value *X* · *T*'Machine_Radix^{-k}, where *k* is the normalized exponent of *X*. A zero result[, which can only occur when *X* is zero,] has the sign of *X*. 23

Discussion: Informally, when *X* is a normalized number, the result is the value obtained by replacing the *exponent* by zero in the canonical-form representation of *X*. 23.a

Ramification: Except when *X* is zero, the magnitude of the result is greater than or equal to the reciprocal of *T*'Machine_Radix and less than one; consequently, the result is always a normalized number, even when *X* is a denormalized number. 23.b

Implementation Note: When *X* is a denormalized number, the result is the value obtained by replacing the *exponent* by zero in the canonical-form representation of the result of scaling *X* up sufficiently to normalize it. 23.c

S'Compose S'Compose denotes a function with the following specification: 24

```
function S'Compose (Fraction : T;
                    Exponent : universal_integer)
  return T
```
25

{*Constraint_Error (raised by failure of run-time check)*} Let *v* be the value *Fraction* · *T*'Machine_Radix$^{Exponent-k}$, where *k* is the normalized exponent of *Fraction*. If *v* is a machine number of the type *T*, or if |*v*| ≥ *T*'Model_Small, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. {*Range_Check* [partial]} {*check,* 26 *language-defined (Range_Check)*} Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *Fraction* when S'Signed_Zeros is True.

26.a **Discussion:** Informally, when *Fraction* and *v* are both normalized numbers, the result is the value obtained by replacing the *exponent* by *Exponent* in the canonical-form representation of *Fraction*.

26.b **Ramification:** If *Exponent* is less than *T*'Machine_Emin and *Fraction* is nonzero, the result is either zero, *T*'Model_Small, or (if *T*'Denorm is True) a denormalized number.

27 S'Scaling S'Scaling denotes a function with the following specification:

28
```
function S'Scaling (X : T;
                    Adjustment : universal_integer)
  return T
```

29 {*Constraint_Error (raised by failure of run-time check)*} Let *v* be the value $X \cdot T\text{'Machine\_Radix}^{Adjustment}$. If *v* is a machine number of the type *T*, or if $|v| \geq T$'Model_Small, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True.

29.a **Discussion:** Informally, when *X* and *v* are both normalized numbers, the result is the value obtained by increasing the *exponent* by *Adjustment* in the canonical-form representation of *X*.

29.b **Ramification:** If *Adjustment* is sufficiently small (i.e., sufficiently negative), the result is either zero, *T*'Model_Small, or (if *T*'Denorm is True) a denormalized number.

30 S'Floor S'Floor denotes a function with the following specification:

31
```
function S'Floor (X : T)
  return T
```

32 The function yields the value $\lfloor X \rfloor$, i.e., the largest (most positive) integral value less than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a positive sign.

33 S'Ceiling S'Ceiling denotes a function with the following specification:

34
```
function S'Ceiling (X : T)
  return T
```

35 The function yields the value $\lceil X \rceil$, i.e., the smallest (most negative) integral value greater than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a negative sign when S'Signed_Zeros is True.

36 S'Rounding S'Rounding denotes a function with the following specification:

37
```
function S'Rounding (X : T)
  return T
```

38 The function yields the integral value nearest to *X*, rounding away from zero if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True.

39 S'Unbiased_Rounding
S'Unbiased_Rounding denotes a function with the following specification:

40
```
function S'Unbiased_Rounding (X : T)
  return T
```

41 The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True.

41.1/2 S'Machine_Rounding
{*AI95-00267-01*} S'Machine_Rounding denotes a function with the following specification:

41.2/2
```
function S'Machine_Rounding (X : T)
  return T
```

The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of *X* when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor.{*unspecified* [partial]} 41.3/2

Discussion: We leave the rounding unspecified, so that users cannot depend on a particular rounding. This attribute is intended for use in cases where the particular rounding chosen is irrelevant. If there is a need to know which way values halfway between two integers are rounded, one of the other rounding attributes should be used. 41.a.1/2

S'Truncation 42

S'Truncation denotes a function with the following specification:

```
function S'Truncation (X : T)
  return T
```
43

The function yields the value $\lceil X \rceil$ when *X* is negative, and $\lfloor X \rfloor$ otherwise. A zero result has the sign of *X* when S'Signed_Zeros is True. 44

S'Remainder 45

S'Remainder denotes a function with the following specification:

```
function S'Remainder (X, Y : T)
  return T
```
46

{*Constraint_Error (raised by failure of run-time check)*} For nonzero *Y*, let *v* be the value *X* – *n* · *Y*, where *n* is the integer nearest to the exact value of *X*/*Y*; if |*n* – *X*/*Y*| = 1/2, then *n* is chosen to be even. If *v* is a machine number of the type *T*, the function yields *v*; otherwise, it yields zero. {*Division_Check* [partial]} {*check, language-defined (Division_Check)*} Constraint_Error is raised if *Y* is zero. A zero result has the sign of *X* when S'Signed_Zeros is True. 47

Ramification: The magnitude of the result is less than or equal to one-half the magnitude of *Y*. 47.a

Discussion: Given machine numbers *X* and *Y* of the type *T*, *v* is necessarily a machine number of the type *T*, except when *Y* is in the neighborhood of zero, *X* is sufficiently close to a multiple of *Y*, and *T*'Denorm is False. 47.b

S'Adjacent S'Adjacent denotes a function with the following specification: 48

```
function S'Adjacent (X, Towards : T)
  return T
```
49

{*Constraint_Error (raised by failure of run-time check)*} If *Towards* = *X*, the function yields *X*; otherwise, it yields the machine number of the type *T* adjacent to *X* in the direction of *Towards*, if that machine number exists. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} If the result would be outside the base range of S, Constraint_Error is raised. When *T*'Signed_Zeros is True, a zero result has the sign of *X*. When *Towards* is zero, its sign has no bearing on the result. 50

Ramification: The value of S'Adjacent(0.0, 1.0) is the smallest normalized positive number of the type *T* when *T*'Denorm is False and the smallest denormalized positive number of the type *T* when *T*'Denorm is True. 50.a

S'Copy_Sign 51

S'Copy_Sign denotes a function with the following specification:

```
function S'Copy_Sign (Value, Sign : T)
  return T
```
52

{*Constraint_Error (raised by failure of run-time check)*} If the value of *Value* is nonzero, the function yields a result whose magnitude is that of *Value* and whose sign is that of *Sign*; otherwise, it yields the value zero. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is optionally raised if the result is outside the base range of S. A zero result has the sign of *Sign* when S'Signed_Zeros is True. 53

Discussion: S'Copy_Sign is provided for convenience in restoring the sign to a quantity from which it has been temporarily removed, or to a related quantity. When S'Signed_Zeros is True, it is also instrumental in determining the sign of a zero quantity, when required. (Because negative and positive zeros compare equal in systems conforming to IEC 559:1989, a negative zero does *not* appear to be negative when compared to zero.) The sign determination is accomplished by transferring the sign of the zero quantity to a nonzero quantity and then testing for a negative result. 53.a

54 S'Leading_Part

S'Leading_Part denotes a function with the following specification:

55
```
function S'Leading_Part (X : T;
                         Radix_Digits : universal_integer)
  return T
```

56 Let v be the value T'Machine_Radix$^{k–Radix\_Digits}$, where k is the normalized exponent of X. The function yields the value

- 57 $\lfloor X/v \rfloor \cdot v$, when X is nonnegative and *Radix_Digits* is positive;
- 58 $\lceil X/v \rceil \cdot v$, when X is negative and *Radix_Digits* is positive.

59 {*Constraint_Error (raised by failure of run-time check)*} {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised when *Radix_Digits* is zero or negative. A zero result[, which can only occur when *X* is zero,] has the sign of *X*.

59.a **Discussion:** Informally, if *X* is nonzero, the result is the value obtained by retaining only the specified number of (leading) significant digits of *X* (in the machine radix), setting all other digits to zero.

59.b **Implementation Note:** The result can be obtained by first scaling *X* up, if necessary to normalize it, then masking the mantissa so as to retain only the specified number of leading digits, then scaling the result back down if *X* was scaled up.

60 S'Machine S'Machine denotes a function with the following specification:

61
```
function S'Machine (X : T)
  return T
```

62 {*Constraint_Error (raised by failure of run-time check)*} If *X* is a machine number of the type *T*, the function yields *X*; otherwise, it yields the value obtained by rounding or truncating *X* to either one of the adjacent machine numbers of the type *T*. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if rounding or truncating *X* to the precision of the machine numbers results in a value outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True.

62.a **Discussion:** All of the primitive function attributes except Rounding and Machine correspond to subprograms in the Generic_Primitive_Functions generic package proposed as a separate ISO standard (ISO/IEC DIS 11729) for Ada 83. The Scaling, Unbiased_Rounding, and Truncation attributes correspond to the Scale, Round, and Truncate functions, respectively, in Generic_Primitive_Functions. The Rounding attribute rounds away from zero; this functionality was not provided in Generic_Primitive_Functions. The name Round was not available for either of the primitive function attributes that perform rounding, since an attribute of that name is used for a different purpose for decimal fixed point types. Likewise, the name Scale was not available, since an attribute of that name is also used for a different purpose for decimal fixed point types. The functionality of the Machine attribute was also not provided in Generic_Primitive_Functions. The functionality of the Decompose procedure of Generic_Primitive_Functions is only provided in the form of the separate attributes Exponent and Fraction. The functionality of the Successor and Predecessor functions of Generic_Primitive_Functions is provided by the extension of the existing Succ and Pred attributes.

62.b **Implementation Note:** The primitive function attributes may be implemented either with appropriate floating point arithmetic operations or with integer and logical operations that act on parts of the representation directly. The latter is strongly encouraged when it is more efficient than the former; it is mandatory when the former cannot deliver the required accuracy due to limitations of the implementation's arithmetic operations.

63 {*model-oriented attributes (of a floating point subtype)*} The following *model-oriented attributes* are defined for any subtype S of a floating point type *T*.

64 S'Model_Mantissa

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to $\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix}) \rceil + 1$, where *d* is the requested decimal precision of *T*, and less than or equal to the value of *T*'Machine_Mantissa. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*.

S'Model_Emin 65

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to the value of *T*'Machine_Emin. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*.

S'Model_Epsilon 66

Yields the value *T*'Machine_Radix$^{1 - T\text{'Model\_Mantissa}}$. The value of this attribute is of the type *universal_real*.

66.a **Discussion:** In most implementations, this attribute yields the absolute value of the difference between one and the smallest machine number of the type *T* above one which, when added to one, yields a machine number different from one. Further discussion can be found in G.2.2.

S'Model_Small 67

Yields the value *T*'Machine_Radix$^{T\text{'Model\_Emin} - 1}$. The value of this attribute is of the type *universal_real*.

67.a **Discussion:** In most implementations, this attribute yields the smallest positive normalized number of the type *T*, i.e. the number corresponding to the positive underflow threshold. In some implementations employing a radix-complement representation for the type *T*, the positive underflow threshold is closer to zero than is the negative underflow threshold, with the consequence that the smallest positive normalized number does not coincide with the positive underflow threshold (i.e., it exceeds the latter). Further discussion can be found in G.2.2.

S'Model S'Model denotes a function with the following specification: 68

69
```
function S'Model (X : T)
  return T
```

70 If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex.

S'Safe_First 71

Yields the lower bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*.

S'Safe_Last 72

Yields the upper bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*.

72.a **Discussion:** A predefined floating point arithmetic operation that yields a value in the safe range of its result type is guaranteed not to overflow.

72.b **To be honest:** An exception is made for exponentiation by a negative exponent in 4.5.6.

72.c **Implementation defined:** The values of the Model_Mantissa, Model_Emin, Model_Epsilon, Model, Safe_First, and Safe_Last attributes, if the Numerics Annex is not supported.

Incompatibilities With Ada 83

72.d {*incompatibilities with Ada 83*} The Epsilon and Mantissa attributes of floating point types are removed from the language and replaced by Model_Epsilon and Model_Mantissa, which may have different values (as a result of changes in the definition of model numbers); the replacement of one set of attributes by another is intended to convert what would be an inconsistent change into an incompatible change.

72.e The Emax, Small, Large, Safe_Emax, Safe_Small, and Safe_Large attributes of floating point types are removed from the language. Small and Safe_Small are collectively replaced by Model_Small, which is functionally equivalent to Safe_Small, though it may have a slightly different value. The others are collectively replaced by Safe_First and Safe_Last. Safe_Last is functionally equivalent to Safe_Large, though it may have a different value; Safe_First is comparable to the negation of Safe_Large but may differ slightly from it as well as from the negation of Safe_Last.

Emax and Safe_Emax had relatively few uses in Ada 83; T'Safe_Emax can be computed in the revised language as Integer'Min(T'Exponent(T'Safe_First), T'Exponent(T'Safe_Last)).

72.f Implementations are encouraged to eliminate the incompatibilities discussed here by retaining the old attributes, during a transition period, in the form of implementation-defined attributes with their former values.

Extensions to Ada 83

72.g {*extensions to Ada 83*} The Model_Emin attribute is new. It is conceptually similar to the negation of Safe_Emax attribute of Ada 83, adjusted for the fact that the model numbers now have the hardware radix. It is a fundamental determinant, along with Model_Mantissa, of the set of model numbers of a type (see G.2.1).

72.h The Denorm and Signed_Zeros attributes are new, as are all of the primitive function attributes.

Extensions to Ada 95

72.i/2 {*AI95-00388-01*} {*extensions to Ada 95*} The Machine_Rounding attribute is new.

A.5.4 Attributes of Fixed Point Types

Static Semantics

1 {*representation-oriented attributes (of a fixed point subtype)*} The following *representation-oriented* attributes are defined for every subtype S of a fixed point type *T*.

2 S'Machine_Radix
: Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*.

3 S'Machine_Rounds
: Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

4 S'Machine_Overflows
: Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

Incompatibilities With Ada 83

4.a {*incompatibilities with Ada 83*} The Mantissa, Large, Safe_Small, and Safe_Large attributes of fixed point types are removed from the language.

4.b Implementations are encouraged to eliminate the resulting incompatibility by retaining these attributes, during a transition period, in the form of implementation-defined attributes with their former values.

Extensions to Ada 83

4.c {*extensions to Ada 83*} The Machine_Radix attribute is now allowed for fixed point types. It is also specifiable in an attribute definition clause (see F.1).

A.6 Input-Output

1/2 {*AI95-00285-01*} [{*input*} {*output*} Input-output is provided through language-defined packages, each of which is a child of the root package Ada. The generic packages Sequential_IO and Direct_IO define input-output operations applicable to files containing elements of a given type. The generic package Storage_IO supports reading from and writing to an in-memory buffer. Additional operations for text input-output are supplied in the packages Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. Heterogeneous input-output is provided through the child packages Streams.Stream_IO and Text_IO.-Text_Streams (see also 13.13). The package IO_Exceptions defines the exceptions needed by the predefined input-output packages.]

Inconsistencies With Ada 83

{*inconsistencies with Ada 83*} The introduction of Append_File as a new element of the enumeration type File_Mode in Sequential_IO and Text_IO, and the introduction of several new declarations in Text_IO, may result in name clashes in the presence of **use** clauses. 1.a

Extensions to Ada 83

{*extensions to Ada 83*} Text_IO enhancements (Get_Immediate, Look_Ahead, Standard_Error, Modular_IO, Decimal_IO), Wide_Text_IO, and the stream input-output facilities are new in Ada 95. 1.b

Wording Changes from Ada 83

RM83-14.6, "Low Level Input-Output," is removed. This has no semantic effect, since the package was entirely implementation defined, nobody actually implemented it, and if they did, they can always provide it as a vendor-supplied package. 1.c

Wording Changes from Ada 95

{*AI95-00285-01*} Included package Wide_Wide_Text_IO in this description. 1.d/2

A.7 External Files and File Objects

Static Semantics

{*external file*} {*name (of an external file)*} {*form (of an external file)*} Values input from the external environment of the program, or output to the external environment, are considered to occupy *external files*. An external file can be anything external to the program that can produce a value to be read or receive a value to be written. An external file is identified by a string (the *name*). A second string (the *form*) gives further system-dependent characteristics that may be associated with the file, such as the physical organization or access rights. The conventions governing the interpretation of such strings shall be documented. 1

{*file (as file object)*} Input and output operations are expressed as operations on objects of some *file type*, rather than directly in terms of the external files. In the remainder of this section, the term *file* is always used to refer to a file object; the term *external file* is used otherwise. 2

Input-output for sequential files of values of a single element type is defined by means of the generic package Sequential_IO. In order to define sequential input-output for a given element type, an instantiation of this generic unit, with the given type as actual parameter, has to be declared. The resulting package contains the declaration of a file type (called File_Type) for files of such elements, as well as the operations applicable to these files, such as the Open, Read, and Write procedures. 3

{*AI95-00285-01*} Input-output for direct access files is likewise defined by a generic package called Direct_IO. Input-output in human-readable form is defined by the (nongeneric) packages Text_IO for Character and String data, Wide_Text_IO for Wide_Character and Wide_String data, and Wide_Wide_Text_IO for Wide_Wide_Character and Wide_Wide_String data. Input-output for files containing streams of elements representing values of possibly different types is defined by means of the (nongeneric) package Streams.Stream_IO. 4/2

Before input or output operations can be performed on a file, the file first has to be associated with an external file. While such an association is in effect, the file is said to be *open*, and otherwise the file is said to be *closed*. 5

The language does not define what happens to external files after the completion of the main program and all the library tasks (in particular, if corresponding files have not been closed). {*access types (input-output unspecified)*} {*input-output (unspecified for access types)*} {*unspecified* [partial]} The effect of input-output for access types is unspecified. 6

7 {*current mode (of an open file)*} An open file has a *current mode*, which is a value of one of the following enumeration types:

8
```
    type File_Mode is (In_File, Inout_File, Out_File);  -- for Direct_IO
```

9 These values correspond respectively to the cases where only reading, both reading and writing, or only writing are to be performed.

10/2
```
    {AI95-00285-01} type File_Mode is (In_File, Out_File, Append_File);
    --  for Sequential_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Stream_IO
```

11 These values correspond respectively to the cases where only reading, only writing, or only appending are to be performed.

12 The mode of a file can be changed.

13/2 {*AI95-00285-01*} Several file management operations are common to Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. These operations are described in subclause A.8.2 for sequential and direct files. Any additional effects concerning text input-output are described in subclause A.10.2.

14 The exceptions that can be propagated by the execution of an input-output subprogram are defined in the package IO_Exceptions; the situations in which they can be propagated are described following the description of the subprogram (and in clause A.13). {*Storage_Error (raised by failure of run-time check)*} {*Program_Error (raised by failure of run-time check)*} The exceptions Storage_Error and Program_Error may be propagated. (Program_Error can only be propagated due to errors made by the caller of the subprogram.) Finally, exceptions can be propagated in certain implementation-defined situations.

14.a/2 *This paragraph was deleted.*

14.b/2 **Discussion:** The last sentence here is referring to the documentation requirements in A.13, “Exceptions in Input-Output”, and the documentation summary item is provided there.

NOTES

15/2 18 {*AI95-00285-01*} Each instantiation of the generic packages Sequential_IO and Direct_IO declares a different type File_Type. In the case of Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Streams.Stream_IO, the corresponding type File_Type is unique.

16 19 A bidirectional device can often be modeled as two sequential files associated with the device, one of mode In_File, and one of mode Out_File. An implementation may restrict the number of files that may be associated with a given external file.

Wording Changes from Ada 95

16.a/2 {*AI95-00285-01*} Included package Wide_Wide_Text_IO in this description.

A.8 Sequential and Direct Files

Static Semantics

1/2 {*AI95-00283-01*} {*sequential file*} {*direct file*} {*stream file*} Two kinds of access to external files are defined in this subclause: *sequential access* and *direct access*. The corresponding file types and the associated operations are provided by the generic packages Sequential_IO and Direct_IO. A file object to be used for sequential access is called a *sequential file*, and one to be used for direct access is called a *direct file*. Access to *stream file*s is described in A.12.1.

2 {*sequential access*} For sequential access, the file is viewed as a sequence of values that are transferred in the order of their appearance (as produced by the program or by the external environment). When the file is opened with mode In_File or Out_File, transfer starts respectively from or to the beginning of the file. When the file is opened with mode Append_File, transfer to the file starts after the last element of the file.

2.a **Discussion:** Adding stream I/O necessitates a review of the terminology. In Ada 83, `sequential' implies both the access method (purely sequential — that is, no indexing or positional access) and homogeneity. Direct access includes purely sequential access and indexed access, as well as homogeneity. In Ada 95, streams allow purely sequential access but also positional access to an individual element, and are heterogeneous. We considered generalizing the notion of `sequential file' to include both Sequential_IO and Stream_IO files, but since streams allow positional access it seems misleading to call them sequential files. Or, looked at differently, if the criterion for calling something a sequential file is whether it permits (versus requires) purely sequential access, then one could just as soon regard a Direct_IO file as a sequential file.

It seems better to regard `sequential file' as meaning `only permitting purely sequential access'; hence we have decided to supplement `sequential access' and `direct access' with a third category, informally called `access to streams'. (We decided against the term `stream access' because of possible confusion with the Stream_Access type declared in one of the stream packages.) 2.b

{*direct access*} {*index (of an element of an open direct file)*} {*current size (of an external file)*} For direct access, the file is viewed as a set of elements occupying consecutive positions in linear order; a value can be transferred to or from an element of the file at any selected position. The position of an element is specified by its *index*, which is a number, greater than zero, of the implementation-defined integer type Count. The first element, if any, has index one; the index of the last element, if any, is called the *current size*; the current size is zero if there are no elements. The current size is a property of the external file. 3

{*current index (of an open direct file)*} An open direct file has a *current index*, which is the index that will be used by the next read or write operation. When a direct file is opened, the current index is set to one. The current index of a direct file is a property of a file object, not of an external file. 4

Wording Changes from Ada 95

{*AI95-00283-01*} Italicized "stream file" to clarify that this is another kind of file. 4.a/2

A.8.1 The Generic Package Sequential_IO

Static Semantics

The generic library package Sequential_IO has the following declaration: 1

```
with Ada.IO_Exceptions;                                              2
generic
   type Element_Type(<>) is private;
package Ada.Sequential_IO is

   type File_Type is limited private;                                3

   type File_Mode is (In_File, Out_File, Append_File);               4

   -- File management                                                5

   procedure Create(File : in out File_Type;                         6
                    Mode : in File_Mode := Out_File;
                    Name : in String := "";
                    Form : in String := "");

   procedure Open  (File : in out File_Type;                         7
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");

   procedure Close (File : in out File_Type);                        8
   procedure Delete(File : in out File_Type);
   procedure Reset (File : in out File_Type; Mode : in File_Mode);
   procedure Reset (File : in out File_Type);

   function Mode   (File : in File_Type) return File_Mode;           9
   function Name   (File : in File_Type) return String;
   function Form   (File : in File_Type) return String;

   function Is_Open(File : in File_Type) return Boolean;             10

   -- Input and output operations                                    11

   procedure Read  (File : in File_Type; Item : out Element_Type);   12
   procedure Write (File : in File_Type; Item : in Element_Type);
```

13
```
      function End_Of_File(File : in File_Type) return Boolean;
```
14
```
      -- Exceptions
```
15
```
      Status_Error : exception renames IO_Exceptions.Status_Error;
      Mode_Error   : exception renames IO_Exceptions.Mode_Error;
      Name_Error   : exception renames IO_Exceptions.Name_Error;
      Use_Error    : exception renames IO_Exceptions.Use_Error;
      Device_Error : exception renames IO_Exceptions.Device_Error;
      End_Error    : exception renames IO_Exceptions.End_Error;
      Data_Error   : exception renames IO_Exceptions.Data_Error;
```
16
```
   private
      ... -- not specified by the language
   end Ada.Sequential_IO;
```

17/2 {*AI95-00360-01*} The type File_Type needs finalization (see 7.6) in every instantiation of Sequential_IO.

Incompatibilities With Ada 83

17.a {*incompatibilities with Ada 83*} The new enumeration element Append_File may introduce upward incompatibilities. It is possible that a program based on the assumption that File_Mode'Last = Out_File will be illegal (e.g., case statement choice coverage) or execute with a different effect in Ada 95.

17.a.1/2 *This paragraph was deleted.*{*8652/0097*} {*AI95-00115-01*} {*AI95-00344-01*}

Incompatibilities With Ada 95

17.b/2 {*AI95-00360-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** File_Type in an instance of Sequential_IO is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and File_Type does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in File_Type, and thus would be illegal.

A.8.2 File Management

Static Semantics

1 The procedures and functions described in this subclause provide for the control of external files; their declarations are repeated in each of the packages for sequential, direct, text, and stream input-output. For text input-output, the procedures Create, Open, and Reset have additional effects described in subclause A.10.2.

2
```
   procedure Create(File : in out File_Type;
                    Mode : in File_Mode := default_mode;
                    Name : in String := "";
                    Form : in String := "");
```

3/2 {*AI95-00283-01*} Establishes a new external file, with the given name and form, and associates this external file with the given file. The given file is left open. The current mode of the given file is set to the given access mode. The default access mode is the mode Out_File for sequential, stream, and text input-output; it is the mode Inout_File for direct input-output. For direct access, the size of the created file is implementation defined.

4 A null string for Name specifies an external file that is not accessible after the completion of the main program (a temporary file). A null string for Form specifies the use of the default options of the implementation for the external file.

5 The exception Status_Error is propagated if the given file is already open. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if, for the specified mode, the external environment does not support creation of an external file with the given name (in the absence of Name_Error) and form.

```
procedure Open(File : in out File_Type;
               Mode : in File_Mode;
               Name : in String;
               Form : in String := "");
```
6

Associates the given file with an existing external file having the given name and form, and sets the current mode of the given file to the given mode. The given file is left open. 7

The exception Status_Error is propagated if the given file is already open. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file; in particular, this exception is propagated if no external file with the given name exists. The exception Use_Error is propagated if, for the specified mode, the external environment does not support opening for an external file with the given name (in the absence of Name_Error) and form. 8

```
procedure Close(File : in out File_Type);
```
9

Severs the association between the given file and its associated external file. The given file is left closed. In addition, for sequential files, if the file being closed has mode Out_File or Append_File, then the last element written since the most recent open or reset is the last element that can be read from the file. If no elements have been written and the file mode is Out_File, then the closed file is empty. If no elements have been written and the file mode is Append_File, then the closed file is unchanged. 10

The exception Status_Error is propagated if the given file is not open. 11

```
procedure Delete(File : in out File_Type);
```
12

Deletes the external file associated with the given file. The given file is closed, and the external file ceases to exist. 13

The exception Status_Error is propagated if the given file is not open. The exception Use_Error is propagated if deletion of the external file is not supported by the external environment. 14

```
procedure Reset(File : in out File_Type; Mode : in File_Mode);
procedure Reset(File : in out File_Type);
```
15

{*AI95-00085-01*} Resets the given file so that reading from its elements can be restarted from the beginning of the external file (for modes In_File and Inout_File), and so that writing to its elements can be restarted at the beginning of the external file (for modes Out_File and Inout_File) or after the last element of the external file (for mode Append_File). In particular, for direct access this means that the current index is set to one. If a Mode parameter is supplied, the current mode of the given file is set to the given mode. In addition, for sequential files, if the given file has mode Out_File or Append_File when Reset is called, the last element written since the most recent open or reset is the last element that can be read from the external file. If no elements have been written and the file mode is Out_File, the reset file is empty. If no elements have been written and the file mode is Append_File, then the reset file is unchanged. 16/2

The exception Status_Error is propagated if the file is not open. The exception Use_Error is propagated if the external environment does not support resetting for the external file and, also, if the external environment does not support resetting to the specified mode for the external file. 17

```
function Mode(File : in File_Type) return File_Mode;
```
18

Returns the current mode of the given file. 19

The exception Status_Error is propagated if the file is not open. 20

21
```
function Name(File : in File_Type) return String;
```

22/2 {*AI95-00248-01*} Returns a string which uniquely identifies the external file currently associated with the given file (and may thus be used in an Open operation).

22.a/2 **Discussion:** {*AI95-00248-01*} Retrieving the full path can be accomplished by passing the result of Name to Directories.Full_Name (see A.16). It is important to drop the requirement on Name, as the only way to accomplish this requirement given that the current directory can be changed with package Directories is to store the full path when the file is opened. That's expensive, and it's better for users that need the full path to explicitly request it.

23 The exception Status_Error is propagated if the given file is not open. The exception Use_Error is propagated if the associated external file is a temporary file that cannot be opened by any name.

24
```
function Form(File : in File_Type) return String;
```

25 Returns the form string for the external file currently associated with the given file. If an external environment allows alternative specifications of the form (for example, abbreviations using default options), the string returned by the function should correspond to a full specification (that is, it should indicate explicitly all options selected, including default options).

26 The exception Status_Error is propagated if the given file is not open.

27
```
function Is_Open(File : in File_Type) return Boolean;
```

28 Returns True if the file is open (that is, if it is associated with an external file), otherwise returns False.

Implementation Permissions

29 An implementation may propagate Name_Error or Use_Error if an attempt is made to use an I/O feature that cannot be supported by the implementation due to limitations in the external environment. Any such restriction should be documented.

Wording Changes from Ada 95

29.a/2 {*AI95-00085-01*} Clarified that Reset affects and depends on the external file.

29.b/2 {*AI95-00248-01*} Removed the requirement for Name to return a full path; this is now accomplished by Directories.Full_Name(Name(File)) (see A.16). This is not documented as an inconsistency, because there is no requirement for implementations to change — the Ada 95 behavior is still allowed, it just is no longer required.

29.c/2 {*AI95-00283-01*} Added text to specify the default mode for a stream file.

A.8.3 Sequential Input-Output Operations

Static Semantics

1 The operations available for sequential input and output are described in this subclause. The exception Status_Error is propagated if any of these operations is attempted for a file that is not open.

2
```
procedure Read(File : in File_Type; Item : out Element_Type);
```

3 Operates on a file of mode In_File. Reads an element from the given file, and returns the value of this element in the Item parameter.

3.a **Discussion:** We considered basing Sequential_IO.Read on Element_Type'Read from an implicit stream associated with the sequential file. However, Element_Type'Read is a type-related attribute, whereas Sequential_IO should take advantage of the particular constraints of the actual subtype corresponding to Element_Type to minimize the size of the external file. Furthermore, forcing the implementation of Sequential_IO to be based on Element_Type'Read would create an upward incompatibility since existing data files written by an Ada 83 program using Sequential_IO might not be readable by the identical program built with an Ada 95 implementation of Sequential_IO.

3.b An Ada 95 implementation might still use an implementation-defined attribute analogous to 'Read to implement the procedure Read, but that attribute will likely have to be subtype-specific rather than type-related, and it need not be user-specifiable. Such an attribute will presumably be needed to implement the generic package Storage_IO (see A.9).

The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if no more elements can be read from the given file. The exception Data_Error can be propagated if the element read cannot be interpreted as a value of the subtype Element_Type (see A.13, "Exceptions in Input-Output"). 4

Discussion: Data_Error need not be propagated if the check is too complex. See A.13, "Exceptions in Input-Output". 4.a

```
procedure Write(File : in File_Type; Item : in Element_Type);
```
5

Operates on a file of mode Out_File or Append_File. Writes the value of Item to the given file. 6

The exception Mode_Error is propagated if the mode is not Out_File or Append_File. The exception Use_Error is propagated if the capacity of the external file is exceeded. 7

```
function End_Of_File(File : in File_Type) return Boolean;
```
8

Operates on a file of mode In_File. Returns True if no more elements can be read from the given file; otherwise returns False. 9

The exception Mode_Error is propagated if the mode is not In_File. 10

A.8.4 The Generic Package Direct_IO

Static Semantics

The generic library package Direct_IO has the following declaration: 1

```
with Ada.IO_Exceptions;                                                   2
generic
   type Element_Type is private;
package Ada.Direct_IO is

   type File_Type is limited private;                                     3

   type File_Mode is (In_File, Inout_File, Out_File);                     4
   type Count     is range 0 .. implementation-defined;
   subtype Positive_Count is Count range 1 .. Count'Last;

   -- File management                                                     5

   procedure Create(File : in out File_Type;                              6
                    Mode : in File_Mode := Inout_File;
                    Name : in String := "";
                    Form : in String := "");

   procedure Open  (File : in out File_Type;                              7
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");

   procedure Close (File : in out File_Type);                             8
   procedure Delete(File : in out File_Type);
   procedure Reset (File : in out File_Type; Mode : in File_Mode);
   procedure Reset (File : in out File_Type);

   function Mode   (File : in File_Type) return File_Mode;                9
   function Name   (File : in File_Type) return String;
   function Form   (File : in File_Type) return String;

   function Is_Open(File : in File_Type) return Boolean;                  10

   -- Input and output operations                                         11

   procedure Read (File : in File_Type; Item : out Element_Type;          12
                                        From : in Positive_Count);
   procedure Read (File : in File_Type; Item : out Element_Type);

   procedure Write(File : in File_Type; Item : in  Element_Type;          13
                                        To   : in Positive_Count);
   procedure Write(File : in File_Type; Item : in Element_Type);
```

14
```
        procedure Set_Index(File : in File_Type; To : in Positive_Count);
```
15
```
        function Index(File : in File_Type) return Positive_Count;
        function Size (File : in File_Type) return Count;
```
16
```
        function End_Of_File(File : in File_Type) return Boolean;
```
17
```
        -- Exceptions
```
18
```
        Status_Error : exception renames IO_Exceptions.Status_Error;
        Mode_Error   : exception renames IO_Exceptions.Mode_Error;
        Name_Error   : exception renames IO_Exceptions.Name_Error;
        Use_Error    : exception renames IO_Exceptions.Use_Error;
        Device_Error : exception renames IO_Exceptions.Device_Error;
        End_Error    : exception renames IO_Exceptions.End_Error;
        Data_Error   : exception renames IO_Exceptions.Data_Error;
```
19
```
    private
       ... -- not specified by the language
    end Ada.Direct_IO;
```
19.a **Reason:** The Element_Type formal of Direct_IO does not have an unknown_discriminant_part (unlike Sequential_IO) so that the implementation can make use of the ability to declare uninitialized variables of the type.

20/2 {*AI95-00360-01*} The type File_Type needs finalization (see 7.6) in every instantiation of Direct_IO.

20.a.1/2 *This paragraph was deleted.*{*8652/0097*} {*AI95-00115-01*} {*AI95-00344-01*}

Incompatibilities With Ada 95

20.a/2 {*AI95-00360-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** File_Type in an instance of Direct_IO is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and File_Type does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in File_Type, and thus would be illegal.

A.8.5 Direct Input-Output Operations

Static Semantics

1 The operations available for direct input and output are described in this subclause. The exception Status_Error is propagated if any of these operations is attempted for a file that is not open.

2
```
    procedure Read(File : in File_Type; Item : out Element_Type;
                                        From : in  Positive_Count);
    procedure Read(File : in File_Type; Item : out Element_Type);
```
3 Operates on a file of mode In_File or Inout_File. In the case of the first form, sets the current index of the given file to the index value given by the parameter From. Then (for both forms) returns, in the parameter Item, the value of the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

4 The exception Mode_Error is propagated if the mode of the given file is Out_File. The exception End_Error is propagated if the index to be used exceeds the size of the external file. The exception Data_Error can be propagated if the element read cannot be interpreted as a value of the subtype Element_Type (see A.13).

5
```
    procedure Write(File : in File_Type; Item : in Element_Type;
                                         To   : in Positive_Count);
    procedure Write(File : in File_Type; Item : in Element_Type);
```
6 Operates on a file of mode Inout_File or Out_File. In the case of the first form, sets the index of the given file to the index value given by the parameter To. Then (for both forms) gives the value of the parameter Item to the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

The exception Mode_Error is propagated if the mode of the given file is In_File. The exception Use_Error is propagated if the capacity of the external file is exceeded. 7

```
procedure Set_Index(File : in File_Type; To : in Positive_Count);
```
8

Operates on a file of any mode. Sets the current index of the given file to the given index value (which may exceed the current size of the file). 9

```
function Index(File : in File_Type) return Positive_Count;
```
10

Operates on a file of any mode. Returns the current index of the given file. 11

```
function Size(File : in File_Type) return Count;
```
12

Operates on a file of any mode. Returns the current size of the external file that is associated with the given file. 13

```
function End_Of_File(File : in File_Type) return Boolean;
```
14

Operates on a file of mode In_File or Inout_File. Returns True if the current index exceeds the size of the external file; otherwise returns False. 15

The exception Mode_Error is propagated if the mode of the given file is Out_File. 16

NOTES

20 Append_File mode is not supported for the generic package Direct_IO. 17

A.9 The Generic Package Storage_IO

The generic package Storage_IO provides for reading from and writing to an in-memory buffer. This generic package supports the construction of user-defined input-output packages. 1

Reason: This package exists to allow the portable construction of user-defined direct-access-oriented input-output packages. The Write procedure writes a value of type Element_Type into a Storage_Array of size Buffer_Size, flattening out any implicit levels of indirection used in the representation of the type. The Read procedure reads a value of type Element_Type from the buffer, reconstructing any implicit levels of indirection used in the representation of the type. It also properly initializes any type tags that appear within the value, presuming that the buffer was written by a different program and that tag values for the“same” type might vary from one executable to another. 1.a

Static Semantics

The generic library package Storage_IO has the following declaration: 2

```
3
with Ada.IO_Exceptions;
with System.Storage_Elements;
generic
   type Element_Type is private;
package Ada.Storage_IO is
   pragma Preelaborate(Storage_IO);

4
   Buffer_Size : constant System.Storage_Elements.Storage_Count :=
      implementation-defined;
   subtype Buffer_Type is
      System.Storage_Elements.Storage_Array(1..Buffer_Size);

5
   -- Input and output operations

6
   procedure Read (Buffer : in  Buffer_Type; Item : out Element_Type);

7
   procedure Write(Buffer : out Buffer_Type; Item : in  Element_Type);

8
   -- Exceptions

9
   Data_Error   : exception renames IO_Exceptions.Data_Error;
end Ada.Storage_IO;
```

In each instance, the constant Buffer_Size has a value that is the size (in storage elements) of the buffer required to represent the content of an object of subtype Element_Type, including any implicit levels of 10 indirection used by the implementation. The Read and Write procedures of Storage_IO correspond to the Read and Write procedures of Direct_IO (see A.8.4), but with the content of the Item parameter being read from or written into the specified Buffer, rather than an external file.

10.a **Reason:** As with Direct_IO, the Element_Type formal of Storage_IO does not have an unknown_discriminant_part so that there is a well-defined upper bound on the size of the buffer needed to hold the content of an object of the formal subtype (i.e. Buffer_Size). If there are no implicit levels of indirection, Buffer_Size will typically equal:

10.b
```
    (Element_Type'Size + System.Storage_Unit - 1) / System.Storage_Unit
```

10.c **Implementation defined:** The value of Buffer_Size in Storage_IO.

NOTES

11 21 A buffer used for Storage_IO holds only one element at a time; an external file used for Direct_IO holds a sequence of elements.

A.10 Text Input-Output

Static Semantics

1 This clause describes the package Text_IO, which provides facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of characters grouped into lines, and as a sequence of lines grouped into pages. The specification of the package is given below in subclause A.10.1.

2 The facilities for file management given above, in subclauses A.8.2 and A.8.3, are available for text input-output. In place of Read and Write, however, there are procedures Get and Put that input values of suitable types from text files, and output values to them. These values are provided to the Put procedures, and returned by the Get procedures, in a parameter Item. Several overloaded procedures of these names exist, for different types of Item. These Get procedures analyze the input sequences of characters based on lexical elements (see Section 2) and return the corresponding values; the Put procedures output the given values as appropriate lexical elements. Procedures Get and Put are also available that input and output individual characters treated as character values rather than as lexical elements. Related to character input are procedures to look ahead at the next character without reading it, and to read a character “immediately” without waiting for an end-of-line to signal availability.

3 In addition to the procedures Get and Put for numeric and enumeration types of Item that operate on text files, analogous procedures are provided that read from and write to a parameter of type String. These procedures perform the same analysis and composition of character sequences as their counterparts which have a file parameter.

4 For all Get and Put procedures that operate on text files, and for many other subprograms, there are forms with and without a file parameter. Each such Get procedure operates on an input file, and each such Put procedure operates on an output file. If no file is specified, a default input file or a default output file is used.

5 {*standard input file*} {*standard output file*} At the beginning of program execution the default input and output files are the so-called standard input file and standard output file. These files are open, have respectively the current modes In_File and Out_File, and are associated with two implementation-defined external files. Procedures are provided to change the current default input file and the current default output file.

5.a/2 **Implementation defined:** The external files associated with the standard input, standard output, and standard error files.

5.a.1/1 **Implementation Note:** {*8652/0113*} {*AI95-00087-01*} The default input file and default output file are not the names of distinct file objects, but rather the *role* played by one or more (other) file object(s). Thus, they generally will be implemented as accesses to another file object. An implementation that implements them by copying them is incorrect.

6 {*standard error file*} At the beginning of program execution a default file for program-dependent error-related text output is the so-called standard error file. This file is open, has the current mode Out_File, and is associated with an implementation-defined external file. A procedure is provided to change the current default error file.

7 {*line terminator*} {*page terminator*} {*file terminator*} From a logical point of view, a text file is a sequence of pages, a page is a sequence of lines, and a line is a sequence of characters; the end of a line is marked by a *line terminator*; the end of a page is marked by the combination of a line terminator immediately followed by a *page terminator*; and the end of a file is marked by the combination of a line terminator immediately followed by a page terminator and then a *file terminator*. Terminators are generated during output; either by calls of procedures provided expressly for that purpose; or implicitly as part of other operations, for example, when a bounded line length, a bounded page length, or both, have been specified for a file.

8 The actual nature of terminators is not defined by the language and hence depends on the implementation. Although terminators are recognized or generated by certain of the procedures that follow, they are not necessarily implemented as characters or as sequences of characters. Whether they are characters (and if so which ones) in any particular implementation need not concern a user who neither explicitly outputs nor explicitly inputs control characters. The effect of input (Get) or output (Put) of control characters (other than horizontal tabulation) is not specified by the language. {*unspecified* [partial]}

9 {*column number*} {*current column number*} {*current line number*} {*current page number*} The characters of a line are numbered, starting from one; the number of a character is called its *column number*. For a line terminator, a column number is also defined: it is one more than the number of characters in the line. The lines of a page, and the pages of a file, are similarly numbered. The current column number is the column number of the next character or line terminator to be transferred. The current line number is the number of the current line. The current page number is the number of the current page. These numbers are values of the subtype Positive_Count of the type Count (by convention, the value zero of the type Count is used to indicate special conditions).

10
```
type Count is range 0 .. implementation-defined;
subtype Positive_Count is Count range 1 .. Count'Last;
```

11 {*maximum line length*} {*maximum page length*} For an output file or an append file, a *maximum line length* can be specified and a *maximum page length* can be specified. If a value to be output cannot fit on the current line, for a specified maximum line length, then a new line is automatically started before the value is output; if, further, this new line cannot fit on the current page, for a specified maximum page length, then a new page is automatically started before the value is output. Functions are provided to determine the maximum line length and the maximum page length. When a file is opened with mode Out_File or Append_File, both values are zero: by convention, this means that the line lengths and page lengths are unbounded. (Consequently, output consists of a single line if the subprograms for explicit control of line and page structure are not used.) The constant Unbounded is provided for this purpose.

Extensions to Ada 83

11.a {*extensions to Ada 83*} Append_File is new in Ada 95.

A.10.1 The Package Text_IO

Static Semantics

1 The library package Text_IO has the following declaration:

2
```
with Ada.IO_Exceptions;
package Ada.Text_IO is
```

3
```
   type File_Type is limited private;
```

```
4      type File_Mode is (In_File, Out_File, Append_File);

5      type Count is range 0 .. implementation-defined;
       subtype Positive_Count is Count range 1 .. Count'Last;
       Unbounded : constant Count := 0; -- line and page length

6      subtype Field       is Integer range 0 .. implementation-defined;
       subtype Number_Base is Integer range 2 .. 16;

7      type Type_Set is (Lower_Case, Upper_Case);

8      -- File Management

9      procedure Create (File : in out File_Type;
                         Mode : in File_Mode := Out_File;
                         Name : in String    := "";
                         Form : in String    := "");

10     procedure Open   (File : in out File_Type;
                         Mode : in File_Mode;
                         Name : in String;
                         Form : in String := "");

11     procedure Close  (File : in out File_Type);
       procedure Delete (File : in out File_Type);
       procedure Reset  (File : in out File_Type; Mode : in File_Mode);
       procedure Reset  (File : in out File_Type);

12     function  Mode   (File : in File_Type) return File_Mode;
       function  Name   (File : in File_Type) return String;
       function  Form   (File : in File_Type) return String;

13     function  Is_Open(File : in File_Type) return Boolean;

14     -- Control of default input and output files

15     procedure Set_Input (File : in File_Type);
       procedure Set_Output(File : in File_Type);
       procedure Set_Error (File : in File_Type);

16     function Standard_Input  return File_Type;
       function Standard_Output return File_Type;
       function Standard_Error  return File_Type;

17     function Current_Input   return File_Type;
       function Current_Output  return File_Type;
       function Current_Error   return File_Type;

18     type File_Access is access constant File_Type;

19     function Standard_Input  return File_Access;
       function Standard_Output return File_Access;
       function Standard_Error  return File_Access;

20     function Current_Input   return File_Access;
       function Current_Output  return File_Access;
       function Current_Error   return File_Access;

21/1 {8652/0051} {AI95-00057-01} --Buffer control
       procedure Flush (File : in File_Type);
       procedure Flush;

22     -- Specification of line and page lengths

23     procedure Set_Line_Length(File : in File_Type; To : in Count);
       procedure Set_Line_Length(To   : in Count);

24     procedure Set_Page_Length(File : in File_Type; To : in Count);
       procedure Set_Page_Length(To   : in Count);

25     function  Line_Length(File : in File_Type) return Count;
       function  Line_Length return Count;

26     function  Page_Length(File : in File_Type) return Count;
       function  Page_Length return Count;

27     -- Column, Line, and Page Control
```

```
28  procedure New_Line   (File    : in File_Type;
                          Spacing : in Positive_Count := 1);
    procedure New_Line   (Spacing : in Positive_Count := 1);

29  procedure Skip_Line  (File    : in File_Type;
                          Spacing : in Positive_Count := 1);
    procedure Skip_Line  (Spacing : in Positive_Count := 1);

30  function  End_Of_Line(File : in File_Type) return Boolean;
    function  End_Of_Line return Boolean;

31  procedure New_Page   (File : in File_Type);
    procedure New_Page;

32  procedure Skip_Page  (File : in File_Type);
    procedure Skip_Page;

33  function  End_Of_Page(File : in File_Type) return Boolean;
    function  End_Of_Page return Boolean;

34  function  End_Of_File(File : in File_Type) return Boolean;
    function  End_Of_File return Boolean;

35  procedure Set_Col (File : in File_Type; To : in Positive_Count);
    procedure Set_Col (To   : in Positive_Count);

36  procedure Set_Line(File : in File_Type; To : in Positive_Count);
    procedure Set_Line(To   : in Positive_Count);

37  function Col (File : in File_Type) return Positive_Count;
    function Col  return Positive_Count;

38  function Line(File : in File_Type) return Positive_Count;
    function Line return Positive_Count;

39  function Page(File : in File_Type) return Positive_Count;
    function Page return Positive_Count;

40  -- Character Input-Output

41  procedure Get(File : in  File_Type; Item : out Character);
    procedure Get(Item : out Character);

42  procedure Put(File : in  File_Type; Item : in Character);
    procedure Put(Item : in  Character);

43  procedure Look_Ahead (File        : in  File_Type;
                          Item        : out Character;
                          End_Of_Line : out Boolean);
    procedure Look_Ahead (Item        : out Character;
                          End_Of_Line : out Boolean);

44  procedure Get_Immediate(File      : in  File_Type;
                            Item      : out Character);
    procedure Get_Immediate(Item      : out Character);

45  procedure Get_Immediate(File      : in  File_Type;
                            Item      : out Character;
                            Available : out Boolean);
    procedure Get_Immediate(Item      : out Character;
                            Available : out Boolean);

46  -- String Input-Output

47  procedure Get(File : in  File_Type; Item : out String);
    procedure Get(Item : out String);

48  procedure Put(File : in  File_Type; Item : in String);
    procedure Put(Item : in  String);

49  procedure Get_Line(File : in  File_Type;
                       Item : out String;
                       Last : out Natural);
    procedure Get_Line(Item : out String; Last : out Natural);

49.1/2  {AI95-00301-01}    function Get_Line(File : in  File_Type) return String;
    function Get_Line return String;
```

50
```
   procedure Put_Line(File : in  File_Type; Item : in String);
   procedure Put_Line(Item : in  String);
```

51
```
-- Generic packages for Input-Output of Integer Types
```

52
```
   generic
      type Num is range <>;
   package Integer_IO is
```

53
```
      Default_Width : Field := Num'Width;
      Default_Base  : Number_Base := 10;
```

54
```
      procedure Get(File  : in  File_Type;
                    Item  : out Num;
                    Width : in Field := 0);
      procedure Get(Item  : out Num;
                    Width : in  Field := 0);
```

55
```
      procedure Put(File  : in File_Type;
                    Item  : in Num;
                    Width : in Field := Default_Width;
                    Base  : in Number_Base := Default_Base);
      procedure Put(Item  : in Num;
                    Width : in Field := Default_Width;
                    Base  : in Number_Base := Default_Base);
      procedure Get(From : in  String;
                    Item : out Num;
                    Last : out Positive);
      procedure Put(To   : out String;
                    Item : in Num;
                    Base : in Number_Base := Default_Base);
```

56
```
   end Integer_IO;
```

57
```
   generic
      type Num is mod <>;
   package Modular_IO is
```

58
```
      Default_Width : Field := Num'Width;
      Default_Base  : Number_Base := 10;
```

59
```
      procedure Get(File  : in  File_Type;
                    Item  : out Num;
                    Width : in Field := 0);
      procedure Get(Item  : out Num;
                    Width : in  Field := 0);
```

60
```
      procedure Put(File  : in File_Type;
                    Item  : in Num;
                    Width : in Field := Default_Width;
                    Base  : in Number_Base := Default_Base);
      procedure Put(Item  : in Num;
                    Width : in Field := Default_Width;
                    Base  : in Number_Base := Default_Base);
      procedure Get(From : in  String;
                    Item : out Num;
                    Last : out Positive);
      procedure Put(To   : out String;
                    Item : in Num;
                    Base : in Number_Base := Default_Base);
```

61
```
   end Modular_IO;
```

62
```
   -- Generic packages for Input-Output of Real Types
```

63
```
   generic
      type Num is digits <>;
   package Float_IO is
```

64
```
      Default_Fore : Field := 2;
      Default_Aft  : Field := Num'Digits-1;
      Default_Exp  : Field := 3;
```

65
```
   procedure Get(File  : in  File_Type;
                 Item  : out Num;
                 Width : in  Field := 0);
   procedure Get(Item  : out Num;
                 Width : in  Field := 0);
```

66
```
   procedure Put(File : in File_Type;
                 Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
   procedure Put(Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
```

67
```
   procedure Get(From : in String;
                 Item : out Num;
                 Last : out Positive);
   procedure Put(To   : out String;
                 Item : in Num;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
end Float_IO;
```

68
```
generic
   type Num is delta <>;
package Fixed_IO is
```

69
```
   Default_Fore : Field := Num'Fore;
   Default_Aft  : Field := Num'Aft;
   Default_Exp  : Field := 0;
```

70
```
   procedure Get(File  : in  File_Type;
                 Item  : out Num;
                 Width : in  Field := 0);
   procedure Get(Item  : out Num;
                 Width : in  Field := 0);
```

71
```
   procedure Put(File : in File_Type;
                 Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
   procedure Put(Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
```

72
```
   procedure Get(From : in  String;
                 Item : out Num;
                 Last : out Positive);
   procedure Put(To   : out String;
                 Item : in Num;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp);
end Fixed_IO;
```

73
```
generic
   type Num is delta <> digits <>;
package Decimal_IO is
```

74
```
   Default_Fore : Field := Num'Fore;
   Default_Aft  : Field := Num'Aft;
   Default_Exp  : Field := 0;
```

75
```
   procedure Get(File  : in  File_Type;
                 Item  : out Num;
                 Width : in  Field := 0);
   procedure Get(Item  : out Num;
                 Width : in  Field := 0);
```

```
76        procedure Put(File : in File_Type;
                        Item : in Num;
                        Fore : in Field := Default_Fore;
                        Aft  : in Field := Default_Aft;
                        Exp  : in Field := Default_Exp);
          procedure Put(Item : in Num;
                        Fore : in Field := Default_Fore;
                        Aft  : in Field := Default_Aft;
                        Exp  : in Field := Default_Exp);

77        procedure Get(From : in  String;
                        Item : out Num;
                        Last : out Positive);
          procedure Put(To   : out String;
                        Item : in Num;
                        Aft  : in Field := Default_Aft;
                        Exp  : in Field := Default_Exp);
       end Decimal_IO;

78     -- Generic package for Input-Output of Enumeration Types

79     generic
          type Enum is (<>);
       package Enumeration_IO is

80        Default_Width   : Field := 0;
          Default_Setting : Type_Set := Upper_Case;

81        procedure Get(File : in  File_Type;
                        Item : out Enum);
          procedure Get(Item : out Enum);

82        procedure Put(File  : in File_Type;
                        Item  : in Enum;
                        Width : in Field    := Default_Width;
                        Set   : in Type_Set := Default_Setting);
          procedure Put(Item  : in Enum;
                        Width : in Field    := Default_Width;
                        Set   : in Type_Set := Default_Setting);

83        procedure Get(From : in  String;
                        Item : out Enum;
                        Last : out Positive);
          procedure Put(To   : out String;
                        Item : in  Enum;
                        Set  : in  Type_Set := Default_Setting);
       end Enumeration_IO;

84  -- Exceptions

85     Status_Error : exception renames IO_Exceptions.Status_Error;
       Mode_Error   : exception renames IO_Exceptions.Mode_Error;
       Name_Error   : exception renames IO_Exceptions.Name_Error;
       Use_Error    : exception renames IO_Exceptions.Use_Error;
       Device_Error : exception renames IO_Exceptions.Device_Error;
       End_Error    : exception renames IO_Exceptions.End_Error;
       Data_Error   : exception renames IO_Exceptions.Data_Error;
       Layout_Error : exception renames IO_Exceptions.Layout_Error;
    private
       ... -- not specified by the language
    end Ada.Text_IO;
```

86/2 {*AI95-00360-01*} The type File_Type needs finalization (see 7.6).

Incompatibilities With Ada 83

86.a {*incompatibilities with Ada 83*} Append_File is a new element of enumeration type File_Mode.

Extensions to Ada 83

{*extensions to Ada 83*} Get_Immediate, Look_Ahead, the subprograms for dealing with standard error, the type File_Access and its associated subprograms, and the generic packages Modular_IO and Decimal_IO are new in Ada 95. 86.b

Incompatibilities With Ada 95

{*AI95-00360-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** Text_IO.File_Type is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and File_Type does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in File_Type, and thus would be illegal. 86.c/2

Wording Changes from Ada 95

{*8652/0051*} {*AI95-00057-01*} **Corrigendum:** Corrected the parameter mode of Flush; otherwise it could not be used on Standard_Output. 86.d/2

{*AI95-00301-01*} The Text_IO.Get_Line functions are new; they are described in A.10.7, “Input-Output of Characters and Strings”. 86.e/2

A.10.2 Text File Management

Static Semantics

1 The only allowed file modes for text files are the modes In_File, Out_File, and Append_File. The subprograms given in subclause A.8.2 for the control of external files, and the function End_Of_File given in subclause A.8.3 for sequential input-output, are also available for text files. There is also a version of End_Of_File that refers to the current default input file. For text files, the procedures have the following additional effects:

- 2 For the procedures Create and Open: After a file with mode Out_File or Append_File is opened, the page length and line length are unbounded (both have the conventional value zero). After a file (of any mode) is opened, the current column, current line, and current page numbers are set to one. If the mode is Append_File, it is implementation defined whether a page terminator will separate preexisting text in the file from the new text to be written.

  2.a **Reason:** For a file with mode Append_File, although it may seem more sensible for Open to set the current column, line, and page number based on the number of pages in the file, the number of lines on the last page, and the number of columns in the last line, we rejected this approach because of implementation costs; it would require the implementation to scan the file before doing the append, or to do processing that would be equivalent in effect.

  2.b For similar reasons, there is no requirement to erase the last page terminator of the file, nor to insert an explicit page terminator in the case when the final page terminator of a file is represented implicitly by the implementation.

- 3 For the procedure Close: If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator.

- 4 For the procedure Reset: If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator. The current column, line, and page numbers are set to one, and the line and page lengths to Unbounded. If the new mode is Append_File, it is implementation defined whether a page terminator will separate preexisting text in the file from the new text to be written.

  4.a **Reason:** The behavior of Reset should be similar to closing a file and reopening it with the given mode

5 The exception Mode_Error is propagated by the procedure Reset upon an attempt to change the mode of a file that is the current default input file, the current default output file, or the current default error file.

NOTES

6 22 An implementation can define the Form parameter of Create and Open to control effects including the following:

- 7 the interpretation of line and column numbers for an interactive file, and
- 8 the interpretation of text formats in a file created by a foreign program.

A.10.3 Default Input, Output, and Error Files

Static Semantics

1 The following subprograms provide for the control of the particular default files that are used when a file parameter is omitted from a Get, Put, or other operation of text input-output described below, or when application-dependent error-related text is to be output.

2
```
procedure Set_Input(File : in File_Type);
```

3 Operates on a file of mode In_File. Sets the current default input file to File.

4 The exception Status_Error is propagated if the given file is not open. The exception Mode_Error is propagated if the mode of the given file is not In_File.

5
```
procedure Set_Output(File : in File_Type);
procedure Set_Error (File : in File_Type);
```

6 Each operates on a file of mode Out_File or Append_File. Set_Output sets the current default output file to File. Set_Error sets the current default error file to File. The exception Status_Error is propagated if the given file is not open. The exception Mode_Error is propagated if the mode of the given file is not Out_File or Append_File.

7
```
function Standard_Input return File_Type;
function Standard_Input return File_Access;
```

8 Returns the standard input file (see A.10), or an access value designating the standard input file, respectively.

9
```
function Standard_Output return File_Type;
function Standard_Output return File_Access;
```

10 Returns the standard output file (see A.10) or an access value designating the standard output file, respectively.

11
```
function Standard_Error return File_Type;
function Standard_Error return File_Access;
```

12/1 {*8652/0052*} {*AI95-00194-01*} Returns the standard error file (see A.10), or an access value designating the standard error file, respectively.

13 The Form strings implicitly associated with the opening of Standard_Input, Standard_Output, and Standard_Error at the start of program execution are implementation defined.

14
```
function Current_Input return File_Type;
function Current_Input return File_Access;
```

15 Returns the current default input file, or an access value designating the current default input file, respectively.

16
```
function Current_Output return File_Type;
function Current_Output return File_Access;
```

17 Returns the current default output file, or an access value designating the current default output file, respectively.

18
```
function Current_Error return File_Type;
function Current_Error return File_Access;
```

19 Returns the current default error file, or an access value designating the current default error file, respectively.

```
{8652/0051} {AI95-00057-01} procedure Flush (File : in File_Type);        20/1
procedure Flush;
```

21 The effect of Flush is the same as the corresponding subprogram in Streams.Stream_IO (see A.12.1). If File is not explicitly specified, Current_Output is used.

Erroneous Execution

22/1 {*8652/0053*} {*AI95-00063-01*} {*erroneous execution (cause)* [partial]} The execution of a program is erroneous if it invokes an operation on a current default input, default output, or default error file, and if the corresponding file object is closed or no longer exists.

22.a.1/1 **Ramification:** {*8652/0053*} {*AI95-00063-01*} Closing a default file, then setting the default file to another open file before accessing it is not erroneous.

23/1 *This paragraph was deleted.*{*8652/0053*} {*AI95-00063-01*}

NOTES

24 23 The standard input, standard output, and standard error files cannot be opened, closed, reset, or deleted, because the parameter File of the corresponding procedures has the mode **in out**.

25 24 The standard input, standard output, and standard error files are different file objects, but not necessarily different external files.

Wording Changes from Ada 95

25.a/2 {*8652/0051*} {*AI95-00057-01*} **Corrigendum:** Corrected the parameter mode of Flush; otherwise it could not be used on Standard_Output.

25.b/2 {*8652/0052*} {*AI95-00194-01*} **Corrigendum:** Corrected Standard_Error so it refers to the correct file.

25.c/2 {*8652/0053*} {*AI95-00063-01*} **Corrigendum:** Clarified that execution is erroneous only when a closed default file is accessed.

A.10.4 Specification of Line and Page Lengths

Static Semantics

1 The subprograms described in this subclause are concerned with the line and page structure of a file of mode Out_File or Append_File. They operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the current default output file. They provide for output of text with a specified maximum line length or page length. In these cases, line and page terminators are output implicitly and automatically when needed. When line and page lengths are unbounded (that is, when they have the conventional value zero), as in the case of a newly opened file, new lines and new pages are only started when explicitly called for.

2 In all cases, the exception Status_Error is propagated if the file to be used is not open; the exception Mode_Error is propagated if the mode of the file is not Out_File or Append_File.

```
procedure Set_Line_Length(File : in File_Type; To : in Count);        3
procedure Set_Line_Length(To   : in Count);
```

4 Sets the maximum line length of the specified output or append file to the number of characters specified by To. The value zero for To specifies an unbounded line length.

4.a **Ramification:** The setting does not affect the lengths of lines in the existing file, rather it only influences subsequent output operations.

5 The exception Use_Error is propagated if the specified line length is inappropriate for the associated external file.

6
```
procedure Set_Page_Length(File : in File_Type; To : in Count);
procedure Set_Page_Length(To   : in Count);
```

7 Sets the maximum page length of the specified output or append file to the number of lines specified by To. The value zero for To specifies an unbounded page length.

8 The exception Use_Error is propagated if the specified page length is inappropriate for the associated external file.

9
```
function Line_Length(File : in File_Type) return Count;
function Line_Length return Count;
```

10 Returns the maximum line length currently set for the specified output or append file, or zero if the line length is unbounded.

11
```
function Page_Length(File : in File_Type) return Count;
function Page_Length return Count;
```

12 Returns the maximum page length currently set for the specified output or append file, or zero if the page length is unbounded.

A.10.5 Operations on Columns, Lines, and Pages

Static Semantics

1 The subprograms described in this subclause provide for explicit control of line and page structure; they operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the appropriate (input or output) current default file. The exception Status_Error is propagated by any of these subprograms if the file to be used is not open.

2
```
procedure New_Line(File : in File_Type; Spacing : in Positive_Count := 1);
procedure New_Line(Spacing : in Positive_Count := 1);
```

3 Operates on a file of mode Out_File or Append_File.

4 For a Spacing of one: Outputs a line terminator and sets the current column number to one. Then increments the current line number by one, except in the case that the current line number is already greater than or equal to the maximum page length, for a bounded page length; in that case a page terminator is output, the current page number is incremented by one, and the current line number is set to one.

5 For a Spacing greater than one, the above actions are performed Spacing times.

6 The exception Mode_Error is propagated if the mode is not Out_File or Append_File.

7
```
procedure Skip_Line(File  : in File_Type; Spacing : in Positive_Count := 1);
procedure Skip_Line(Spacing : in Positive_Count := 1);
```

8 Operates on a file of mode In_File.

9 For a Spacing of one: Reads and discards all characters until a line terminator has been read, and then sets the current column number to one. If the line terminator is not immediately followed by a page terminator, the current line number is incremented by one. Otherwise, if the line terminator is immediately followed by a page terminator, then the page terminator is skipped, the current page number is incremented by one, and the current line number is set to one.

10 For a Spacing greater than one, the above actions are performed Spacing times.

11 The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if an attempt is made to read a file terminator.

12
```
function End_Of_Line(File : in File_Type) return Boolean;
function End_Of_Line return Boolean;
```

13 Operates on a file of mode In_File. Returns True if a line terminator or a file terminator is next; otherwise returns False.

14 The exception Mode_Error is propagated if the mode is not In_File.

15
```
procedure New_Page(File : in File_Type);
procedure New_Page;
```

16 Operates on a file of mode Out_File or Append_File. Outputs a line terminator if the current line is not terminated, or if the current page is empty (that is, if the current column and line numbers are both equal to one). Then outputs a page terminator, which terminates the current page. Adds one to the current page number and sets the current column and line numbers to one.

17 The exception Mode_Error is propagated if the mode is not Out_File or Append_File.

18
```
procedure Skip_Page(File : in File_Type);
procedure Skip_Page;
```

19 Operates on a file of mode In_File. Reads and discards all characters and line terminators until a page terminator has been read. Then adds one to the current page number, and sets the current column and line numbers to one.

20 The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if an attempt is made to read a file terminator.

21
```
function End_Of_Page(File : in File_Type) return Boolean;
function End_Of_Page return Boolean;
```

22 Operates on a file of mode In_File. Returns True if the combination of a line terminator and a page terminator is next, or if a file terminator is next; otherwise returns False.

23 The exception Mode_Error is propagated if the mode is not In_File.

24
```
function End_Of_File(File : in File_Type) return Boolean;
function End_Of_File return Boolean;
```

25 Operates on a file of mode In_File. Returns True if a file terminator is next, or if the combination of a line, a page, and a file terminator is next; otherwise returns False.

26 The exception Mode_Error is propagated if the mode is not In_File.

27 The following subprograms provide for the control of the current position of reading or writing in a file. In all cases, the default file is the current output file.

28
```
procedure Set_Col(File : in File_Type; To : in Positive_Count);
procedure Set_Col(To   : in Positive_Count);
```

29 If the file mode is Out_File or Append_File:

30
- If the value specified by To is greater than the current column number, outputs spaces, adding one to the current column number after each space, until the current column number equals the specified value. If the value specified by To is equal to the current column number, there is no effect. If the value specified by To is less than the current column number, has the effect of calling New_Line (with a spacing of one), then outputs (To – 1) spaces, and sets the current column number to the specified value.

31
- The exception Layout_Error is propagated if the value specified by To exceeds Line_Length when the line length is bounded (that is, when it does not have the conventional value zero).

32 If the file mode is In_File:

33 • Reads (and discards) individual characters, line terminators, and page terminators, until the next character to be read has a column number that equals the value specified by To; there is no effect if the current column number already equals this value. Each transfer of a character or terminator maintains the current column, line, and page numbers in the same way as a Get procedure (see A.10.6). (Short lines will be skipped until a line is reached that has a character at the specified column position.)

34 • The exception End_Error is propagated if an attempt is made to read a file terminator.

35
```
procedure Set_Line(File : in File_Type; To : in Positive_Count);
procedure Set_Line(To   : in Positive_Count);
```

36 If the file mode is Out_File or Append_File:

37 • If the value specified by To is greater than the current line number, has the effect of repeatedly calling New_Line (with a spacing of one), until the current line number equals the specified value. If the value specified by To is equal to the current line number, there is no effect. If the value specified by To is less than the current line number, has the effect of calling New_Page followed by a call of New_Line with a spacing equal to (To – 1).

38 • The exception Layout_Error is propagated if the value specified by To exceeds Page_Length when the page length is bounded (that is, when it does not have the conventional value zero).

39 If the mode is In_File:

40 • Has the effect of repeatedly calling Skip_Line (with a spacing of one), until the current line number equals the value specified by To; there is no effect if the current line number already equals this value. (Short pages will be skipped until a page is reached that has a line at the specified line position.)

41 • The exception End_Error is propagated if an attempt is made to read a file terminator.

42
```
function Col(File : in File_Type) return Positive_Count;
function Col return Positive_Count;
```

43 Returns the current column number.

44 The exception Layout_Error is propagated if this number exceeds Count'Last.

45
```
function Line(File : in File_Type) return Positive_Count;
function Line return Positive_Count;
```

46 Returns the current line number.

47 The exception Layout_Error is propagated if this number exceeds Count'Last.

48
```
function Page(File : in File_Type) return Positive_Count;
function Page return Positive_Count;
```

49 Returns the current page number.

50 The exception Layout_Error is propagated if this number exceeds Count'Last.

51 The column number, line number, or page number are allowed to exceed Count'Last (as a consequence of the input or output of sufficiently many characters, lines, or pages). These events do not cause any exception to be propagated. However, a call of Col, Line, or Page propagates the exception Layout_Error if the corresponding number exceeds Count'Last.

NOTES

52 25 A page terminator is always skipped whenever the preceding line terminator is skipped. An implementation may represent the combination of these terminators by a single character, provided that it is properly recognized on input.

A.10.6 Get and Put Procedures

Static Semantics

1 The procedures Get and Put for items of the type Character, String, numeric types, and enumeration types are described in subsequent subclauses. Features of these procedures that are common to most of these types are described in this subclause. The Get and Put procedures for items of type Character and String deal with individual character values; the Get and Put procedures for numeric and enumeration types treat the items as lexical elements.

2 All procedures Get and Put have forms with a file parameter, written first. Where this parameter is omitted, the appropriate (input or output) current default file is understood to be specified. Each procedure Get operates on a file of mode In_File. Each procedure Put operates on a file of mode Out_File or Append_File.

3 All procedures Get and Put maintain the current column, line, and page numbers of the specified file: the effect of each of these procedures upon these numbers is the result of the effects of individual transfers of characters and of individual output or skipping of terminators. Each transfer of a character adds one to the current column number. Each output of a line terminator sets the current column number to one and adds one to the current line number. Each output of a page terminator sets the current column and line numbers to one and adds one to the current page number. For input, each skipping of a line terminator sets the current column number to one and adds one to the current line number; each skipping of a page terminator sets the current column and line numbers to one and adds one to the current page number. Similar considerations apply to the procedures Get_Line, Put_Line, and Set_Col.

4 Several Get and Put procedures, for numeric and enumeration types, have *format* parameters which specify field lengths; these parameters are of the nonnegative subtype Field of the type Integer.

5/2 {*AI95-00223-01*} {*blank (in text input for enumeration and numeric types)*} Input-output of enumeration values uses the syntax of the corresponding lexical elements. Any Get procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators. A *blank* is defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input.

6 For a numeric type, the Get procedures have a format parameter called Width. If the value given for this parameter is zero, the Get procedure proceeds in the same manner as for enumeration types, but using the syntax of numeric literals instead of that of enumeration literals. If a nonzero value is given, then exactly Width characters are input, or the characters up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. The syntax used for numeric literals is an extended syntax that allows a leading sign (but no intervening blanks, or line or page terminators) and that also allows (for real types) an integer literal as well as forms that have digits only before the point or only after the point.

7 Any Put procedure, for an item of a numeric or an enumeration type, outputs the value of the item as a numeric literal, identifier, or character literal, as appropriate. This is preceded by leading spaces if required by the format parameters Width or Fore (as described in later subclauses), and then a minus sign for a negative value; for an enumeration type, the spaces follow instead of leading. The format given for a Put procedure is overridden if it is insufficiently wide, by using the minimum needed width.

8 Two further cases arise for Put procedures for numeric and enumeration types, if the line length of the specified output file is bounded (that is, if it does not have the conventional value zero). If the number of characters to be output does not exceed the maximum line length, but is such that they cannot fit on the

current line, starting from the current column, then (in effect) New_Line is called (with a spacing of one) before output of the item. Otherwise, if the number of characters exceeds the maximum line length, then the exception Layout_Error is propagated and nothing is output.

9 The exception Status_Error is propagated by any of the procedures Get, Get_Line, Put, and Put_Line if the file to be used is not open. The exception Mode_Error is propagated by the procedures Get and Get_Line if the mode of the file to be used is not In_File; and by the procedures Put and Put_Line, if the mode is not Out_File or Append_File.

10 The exception End_Error is propagated by a Get procedure if an attempt is made to skip a file terminator. The exception Data_Error is propagated by a Get procedure if the sequence finally input is not a lexical element corresponding to the type, in particular if no characters were input; for this test, leading blanks are ignored; for an item of a numeric type, when a sign is input, this rule applies to the succeeding numeric literal. The exception Layout_Error is propagated by a Put procedure that outputs to a parameter of type String, if the length of the actual string is insufficient for the output of the item.

Examples

11 In the examples, here and in subclauses A.10.8 and A.10.9, the string quotes and the lower case letter b are not transferred: they are shown only to reveal the layout and spaces.

12
```
N : Integer;
   ...
Get(N);
```

13
```
--  Characters at input   Sequence input   Value of N

--  bb–12535b             –12535           –12535
--  bb12_535e1b           12_535e1         125350
--  bb12_535e;            12_535e          (none) Data_Error raised
```

14 Example of overridden width parameter:

15
```
Put(Item => -23, Width => 2);  -- "–23"
```

Wording Changes from Ada 95

15.a/2 {*AI95-00223-01*} Removed conflicting text describing the skipping of blanks for a Get procedure.

A.10.7 Input-Output of Characters and Strings

Static Semantics

1 For an item of type Character the following procedures are provided:

2
```
procedure Get(File : in File_Type; Item : out Character);
procedure Get(Item : out Character);
```

3 After skipping any line terminators and any page terminators, reads the next character from the specified input file and returns the value of this character in the out parameter Item.

4 The exception End_Error is propagated if an attempt is made to skip a file terminator.

5
```
procedure Put(File : in File_Type; Item : in Character);
procedure Put(Item : in Character);
```

6 If the line length of the specified output file is bounded (that is, does not have the conventional value zero), and the current column number exceeds it, has the effect of calling New_Line with a spacing of one. Then, or otherwise, outputs the given character to the file.

7
```
procedure Look_Ahead (File        : in  File_Type;
                      Item        : out Character;
                      End_Of_Line : out Boolean);
procedure Look_Ahead (Item        : out Character;
                      End_Of_Line : out Boolean);
```

8/1 Mode_Error is propagated if the mode of the file is not In_File. Sets End_Of_Line to True if at end of line, including if at end of page or at end of file; in each of these cases the value of Item is not specified. {*unspecified* [partial]} Otherwise End_Of_Line is set to False and Item is set to the next character (without consuming it) from the file.

9
```
procedure Get_Immediate(File : in  File_Type;
                        Item : out Character);
procedure Get_Immediate(Item : out Character);
```

10 Reads the next character, either control or graphic, from the specified File or the default input file. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

11
```
procedure Get_Immediate(File      : in  File_Type;
                        Item      : out Character;
                        Available : out Boolean);
procedure Get_Immediate(Item      : out Character;
                        Available : out Boolean);
```

12 If a character, either control or graphic, is available from the specified File or the default input file, then the character is read; Available is True and Item contains the value of this character. If a character is not available, then Available is False and the value of Item is not specified. {*unspecified* [partial]} Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

13/2 {*AI95-00301-01*} For an item of type String the following subprograms are provided:

14
```
procedure Get(File : in File_Type; Item : out String);
procedure Get(Item : out String);
```

15 Determines the length of the given string and attempts that number of Get operations for successive characters of the string (in particular, no operation is performed if the string is null).

16
```
procedure Put(File : in File_Type; Item : in String);
procedure Put(Item : in String);
```

17 Determines the length of the given string and attempts that number of Put operations for successive characters of the string (in particular, no operation is performed if the string is null).

17.1/2
```
function Get_Line(File : in File_Type) return String;
function Get_Line return String;
```

17.2/2 {*AI95-00301-01*} Returns a result string constructed by reading successive characters from the specified input file, and assigning them to successive characters of the result string. The result string has a lower bound of 1 and an upper bound of the number of characters read. Reading stops when the end of the line is met; Skip_Line is then (in effect) called with a spacing of 1.

17.3/2 {*AI95-00301-01*} Constraint_Error is raised if the length of the line exceeds Positive'Last; in this case, the line number and page number are unchanged, and the column number is unspecified but no less than it was before the call.{*unspecified* [partial]} The exception End_Error is propagated if an attempt is made to skip a file terminator.

17.a/2 **Ramification:** {*AI95-00301-01*} Precisely what is left in the file is unspecified if Constraint_Error is raised because the line doesn't fit in a String; it should be consistent with column number. This allows implementers to use whatever buffering scheme makes sense. But the line terminator is not skipped in this case.

18
```
procedure Get_Line(File : in File_Type;
                            Item : out String;
                            Last : out Natural);
procedure Get_Line(Item : out String;   Last : out Natural);
```

19 Reads successive characters from the specified input file and assigns them to successive characters of the specified string. Reading stops if the end of the string is met. Reading also stops if the end of the line is met before meeting the end of the string; in this case Skip_Line is (in effect) called with a spacing of 1. {*unspecified* [partial]} The values of characters not assigned are not specified.

20 If characters are read, returns in Last the index value such that Item(Last) is the last character assigned (the index of the first character assigned is Item'First). If no characters are read, returns in Last an index value that is one less than Item'First. The exception End_Error is propagated if an attempt is made to skip a file terminator.

21
```
procedure Put_Line(File : in File_Type; Item : in String);
procedure Put_Line(Item : in String);
```

22 Calls the procedure Put for the given string, and then the procedure New_Line with a spacing of one.

Implementation Advice

23 The Get_Immediate procedures should be implemented with unbuffered input. For a device such as a keyboard, input should be “available” if a key has already been typed, whereas for a disk file, input should always be available except at end of file. For a file associated with a keyboard-like device, any line-editing features of the underlying operating system should be disabled during the execution of Get_Immediate.

23.a/2 **Implementation Advice:** Get_Immediate should be implemented with unbuffered input; input should be available immediately; line-editing should be disabled.

NOTES

24 26 Get_Immediate can be used to read a single key from the keyboard “immediately”; that is, without waiting for an end of line. In a call of Get_Immediate without the parameter Available, the caller will wait until a character is available.

25 27 In a literal string parameter of Put, the enclosing string bracket characters are not output. Each doubled string bracket character in the enclosed string is output as a single string bracket character, as a consequence of the rule for string literals (see 2.6).

26 28 A string read by Get or written by Put can extend over several lines. An implementation is allowed to assume that certain external files do not contain page terminators, in which case Get_Line and Skip_Line can return as soon as a line terminator is read.

Incompatibilities With Ada 95

26.a/2 {*AI95-00301-01*} {*incompatibilities with Ada 95*} The Get_Line functions are newly added to Ada.Text_IO. If Ada.Text_IO is referenced in a use_clause, and a function Get_Line is defined in a package that is also referenced in a use_clause, the user-defined Get_Line may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 95

26.b/2 {*AI95-00301-01*} {*extensions to Ada 95*} The Text_IO.Get_Line functions are new.

A.10.8 Input-Output for Integer Types

Static Semantics

1 The following procedures are defined in the generic packages Integer_IO and Modular_IO, which have to be instantiated for the appropriate signed integer or modular type respectively (indicated by Num in the specifications).

2 Values are output as decimal or based literals, without low line characters or exponent, and, for Integer_IO, preceded by a minus sign if negative. The format (which includes any leading spaces and minus sign) can be specified by an optional field width parameter. Values of widths of fields in output formats are of the nonnegative integer subtype Field. Values of bases are of the integer subtype Number_Base.

3
```
subtype Number_Base is Integer range 2 .. 16;
```

4 The default field width and base to be used by output procedures are defined by the following variables that are declared in the generic packages Integer_IO and Modular_IO:

5
```
Default_Width : Field := Num'Width;
Default_Base  : Number_Base := 10;
```

6 The following procedures are provided:

7
```
procedure Get(File : in File_Type; Item : out Num; Width : in Field := 0);
procedure Get(Item : out Num; Width : in Field := 0);
```

8 If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus sign if present or (for a signed type only) a minus sign if present, then reads the longest possible sequence of characters matching the syntax of a numeric literal without a point. If a nonzero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count.

9 Returns, in the parameter Item, the value of type Num that corresponds to the sequence input.

10 The exception Data_Error is propagated if the sequence of characters read does not form a legal integer literal or if the value obtained is not of the subtype Num (for Integer_IO) or is not in the base range of Num (for Modular_IO).

11
```
procedure Put(File  : in File_Type;
              Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);

procedure Put(Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);
```

12 Outputs the value of the parameter Item as an integer literal, with no low lines, no exponent, and no leading zeros (but a single zero for the value zero), and a preceding minus sign for a negative value.

13 If the resulting sequence of characters to be output has fewer than Width characters, then leading spaces are first output to make up the difference.

14 Uses the syntax for decimal literal if the parameter Base has the value ten (either explicitly or through Default_Base); otherwise, uses the syntax for based literal, with any letters in upper case.

15
```
procedure Get(From : in String; Item : out Num; Last : out Positive);
```

16 Reads an integer value from the beginning of the given string, following the same rules as the Get procedure that reads an integer value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

17 The exception Data_Error is propagated if the sequence input does not have the required syntax or if the value obtained is not of the subtype Num.

18
```
procedure Put(To   : out String;
              Item : in Num;
              Base : in Number_Base := Default_Base);
```

19 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using the length of the given string as the value for Width.

20 Integer_Text_IO is a library package that is a nongeneric equivalent to Text_IO.Integer_IO for the predefined type Integer:

21
```
with Ada.Text_IO;
package Ada.Integer_Text_IO is new Ada.Text_IO.Integer_IO(Integer);
```

22 For each predefined signed integer type, a nongeneric equivalent to Text_IO.Integer_IO is provided, with names such as Ada.Long_Integer_Text_IO.

Implementation Permissions

23 The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

NOTES

24 29 For Modular_IO, execution of Get propagates Data_Error if the sequence of characters read forms an integer literal outside the range 0..Num'Last.

Examples

25/1 *This paragraph was deleted.*

26
```
package Int_IO is new Integer_IO(Small_Int); use Int_IO;
-- default format used at instantiation,
-- Default_Width = 4, Default_Base = 10
```

27
```
Put(126);                            -- "b126"
Put(-126, 7);                        -- "bbb–126"
Put(126, Width => 13, Base => 2);    -- "bbb2#1111110#"
```

A.10.9 Input-Output for Real Types

Static Semantics

1 The following procedures are defined in the generic packages Float_IO, Fixed_IO, and Decimal_IO, which have to be instantiated for the appropriate floating point, ordinary fixed point, or decimal fixed point type respectively (indicated by Num in the specifications).

2 Values are output as decimal literals without low line characters. The format of each value output consists of a Fore field, a decimal point, an Aft field, and (if a nonzero Exp parameter is supplied) the letter E and an Exp field. The two possible formats thus correspond to:

3
```
Fore  .  Aft
```

4 and to:

5
```
Fore  .  Aft  E  Exp
```

6 without any spaces between these fields. The Fore field may include leading spaces, and a minus sign for negative values. The Aft field includes only decimal digits (possibly with trailing zeros). The Exp field includes the sign (plus or minus) and the exponent (possibly with leading zeros).

7 For floating point types, the default lengths of these fields are defined by the following variables that are declared in the generic package Float_IO:

8
```
Default_Fore : Field := 2;
Default_Aft  : Field := Num'Digits-1;
Default_Exp  : Field := 3;
```

For ordinary or decimal fixed point types, the default lengths of these fields are defined by the following variables that are declared in the generic packages Fixed_IO and Decimal_IO, respectively: 9

10
```
Default_Fore : Field := Num'Fore;
Default_Aft  : Field := Num'Aft;
Default_Exp  : Field := 0;
```

The following procedures are provided: 11

12
```
procedure Get(File : in File_Type; Item : out Num; Width : in Field := 0);
procedure Get(Item : out Num; Width : in Field := 0);
```

If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads the longest possible sequence of characters matching the syntax of any of the following (see 2.4): 13

- [+|–]numeric_literal 14
- [+|–]numeral.[exponent] 15
- [+|–].numeral[exponent] 16
- [+|–]base#based_numeral.#[exponent] 17
- [+|–]base#.based_numeral#[exponent] 18

If a nonzero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. 19

Returns in the parameter Item the value of type Num that corresponds to the sequence input, preserving the sign (positive if none has been specified) of a zero value if Num is a floating point type and Num'Signed_Zeros is True. 20

The exception Data_Error is propagated if the sequence input does not have the required syntax or if the value obtained is not of the subtype Num. 21

22
```
procedure Put(File : in File_Type;
              Item : in Num;
              Fore : in Field := Default_Fore;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);

procedure Put(Item : in Num;
              Fore : in Field := Default_Fore;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
```

Outputs the value of the parameter Item as a decimal literal with the format defined by Fore, Aft and Exp. If the value is negative, or if Num is a floating point type where Num'Signed_Zeros is True and the value is a negatively signed zero, then a minus sign is included in the integer part. If Exp has the value zero, then the integer part to be output has as many digits as are needed to represent the integer part of the value of Item, overriding Fore if necessary, or consists of the digit zero if the value of Item has no integer part. 23

If Exp has a value greater than zero, then the integer part to be output has a single digit, which is nonzero except for the value 0.0 of Item. 24

In both cases, however, if the integer part to be output has fewer than Fore characters, including any minus sign, then leading spaces are first output to make up the difference. The number of digits of the fractional part is given by Aft, or is one if Aft equals zero. The value is rounded; a value of exactly one half in the last place is rounded away from zero. 25

26 If Exp has the value zero, there is no exponent part. If Exp has a value greater than zero, then the exponent part to be output has as many digits as are needed to represent the exponent part of the value of Item (for which a single digit integer part is used), and includes an initial sign (plus or minus). If the exponent part to be output has fewer than Exp characters, including the sign, then leading zeros precede the digits, to make up the difference. For the value 0.0 of Item, the exponent has the value zero.

27
```
procedure Get(From : in String; Item : out Num; Last : out Positive);
```

28 Reads a real value from the beginning of the given string, following the same rule as the Get procedure that reads a real value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

29 The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the value obtained is not of the subtype Num.

30
```
procedure Put(To   : out String;
              Item : in Num;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
```

31 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using a value for Fore such that the sequence of characters output exactly fills the string, including any leading spaces.

32 Float_Text_IO is a library package that is a nongeneric equivalent to Text_IO.Float_IO for the predefined type Float:

33
```
with Ada.Text_IO;
package Ada.Float_Text_IO is new Ada.Text_IO.Float_IO(Float);
```

34 For each predefined floating point type, a nongeneric equivalent to Text_IO.Float_IO is provided, with names such as Ada.Long_Float_Text_IO.

Implementation Permissions

35 An implementation may extend Get [and Put] for floating point types to support special values such as infinities and NaNs.

35.a **Discussion:** See also the similar permission for the Wide_Value attribute in 3.5.

36 The implementation of Put need not produce an output value with greater accuracy than is supported for the base subtype. The additional accuracy, if any, of the value produced by Put when the number of requested digits in the integer and fractional parts exceeds the required accuracy is implementation defined.

36.a **Discussion:** The required accuracy is thus Num'Base'Digits digits if Num is a floating point subtype. For a fixed point subtype the required accuracy is a function of the subtype's Fore, Aft, and Delta attributes.

36.b **Implementation defined:** The accuracy of the value produced by Put.

37 The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

NOTES

38 30 For an item with a positive value, if output to a string exactly fills the string without leading spaces, then output of the corresponding negative value will propagate Layout_Error.

39 31 The rules for the Value attribute (see 3.5) and the rules for Get are based on the same set of formats.

Examples

This paragraph was deleted. 40/1

41
```
package Real_IO is new Float_IO(Real); use Real_IO;
-- default format used at instantiation, Default_Exp = 3
```

42
```
X : Real := -123.4567;  -- digits 8     (see 3.5.7)
```

43
```
Put(X);  -- default format                        "–1.2345670E+02"
Put(X, Fore => 5, Aft => 3, Exp => 2);    -- "bbb–1.235E+2"
Put(X, 5, 3, 0);                          -- "b–123.457"
```

A.10.10 Input-Output for Enumeration Types

Static Semantics

1 The following procedures are defined in the generic package Enumeration_IO, which has to be instantiated for the appropriate enumeration type (indicated by Enum in the specification).

2 Values are output using either upper or lower case letters for identifiers. This is specified by the parameter Set, which is of the enumeration type Type_Set.

3
```
type Type_Set is (Lower_Case, Upper_Case);
```

4 The format (which includes any trailing spaces) can be specified by an optional field width parameter. The default field width and letter case are defined by the following variables that are declared in the generic package Enumeration_IO:

5
```
Default_Width   : Field := 0;
Default_Setting : Type_Set := Upper_Case;
```

6 The following procedures are provided:

7
```
procedure Get(File : in File_Type; Item : out Enum);
procedure Get(Item : out Enum);
```

8 After skipping any leading blanks, line terminators, or page terminators, reads an identifier according to the syntax of this lexical element (lower and upper case being considered equivalent), or a character literal according to the syntax of this lexical element (including the apostrophes). Returns, in the parameter Item, the value of type Enum that corresponds to the sequence input.

9 The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype Enum.

10
```
procedure Put(File  : in File_Type;
              Item  : in Enum;
              Width : in Field := Default_Width;
              Set   : in Type_Set := Default_Setting);

procedure Put(Item  : in Enum;
              Width : in Field := Default_Width;
              Set   : in Type_Set := Default_Setting);
```

11 Outputs the value of the parameter Item as an enumeration literal (either an identifier or a character literal). The optional parameter Set indicates whether lower case or upper case is used for identifiers; it has no effect for character literals. If the sequence of characters produced has fewer than Width characters, then trailing spaces are finally output to make up the difference. If Enum is a character type, the sequence of characters produced is as for Enum'Image(Item), as modified by the Width and Set parameters.

11.a **Discussion:** For a character type, the literal might be a Wide_Character or a control character. Whatever Image does for these things is appropriate here, too.

12
```
procedure Get(From : in String; Item : out Enum; Last : out Positive);
```

13 Reads an enumeration value from the beginning of the given string, following the same rule as the Get procedure that reads an enumeration value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Enum that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

14 The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype Enum.

14.a **To be honest:** For a character type, it is permissible for the implementation to make Get do the inverse of what Put does, in the case of wide character_literals and control characters.

15
```
procedure Put(To   : out String;
              Item : in Enum;
              Set  : in Type_Set := Default_Setting);
```

16 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using the length of the given string as the value for Width.

17/1 {*8652/0054*} {*AI95-00007-01*} Although the specification of the generic package Enumeration_IO would allow instantiation for an integer type, this is not the intended purpose of this generic package, and the effect of such instantiations is not defined by the language.

NOTES

18 32 There is a difference between Put defined for characters, and for enumeration values. Thus

19
```
Ada.Text_IO.Put('A');  -- outputs the character A
```

20
```
package Char_IO is new Ada.Text_IO.Enumeration_IO(Character);
Char_IO.Put('A');  -- outputs the character 'A', between apostrophes
```

21 33 The type Boolean is an enumeration type, hence Enumeration_IO can be instantiated for this type.

Wording Changes from Ada 95

21.a/2 {*8652/0054*} {*AI95-00007-01*} **Corrigendum:** Corrected the wording to say Enumeration_IO can be instantiated with an integer type, not a float type.

A.10.11 Input-Output for Bounded Strings

1/2 {*AI95-00428-01*} The package Text_IO.Bounded_IO provides input-output in human-readable form for Bounded_Strings.

Static Semantics

2/2 {*AI95-00428-01*} The generic library package Text_IO.Bounded_IO has the following declaration:

3/2
```
with Ada.Strings.Bounded;
generic
   with package Bounded is
                     new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
package Ada.Text_IO.Bounded_IO is
```

4/2
```
   procedure Put
      (File : in File_Type;
       Item : in Bounded.Bounded_String);
```

5/2
```
   procedure Put
      (Item : in Bounded.Bounded_String);
```

6/2
```
   procedure Put_Line
      (File : in File_Type;
       Item : in Bounded.Bounded_String);
```

7/2
```
   procedure Put_Line
      (Item : in Bounded.Bounded_String);
      function Get_Line                                                    8/2
         (File : in File_Type)
         return Bounded.Bounded_String;

      function Get_Line                                                    9/2
         return Bounded.Bounded_String;

      procedure Get_Line                                                   10/2
         (File : in File_Type; Item : out Bounded.Bounded_String);

      procedure Get_Line                                                   11/2
         (Item : out Bounded.Bounded_String);

   end Ada.Text_IO.Bounded_IO;                                             12/2
```

{*AI95-00428-01*} For an item of type Bounded_String, the following subprograms are provided: 13/2

```
   procedure Put                                                           14/2
      (File : in File_Type;
       Item : in Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Text_IO.Put (File, Bounded.To_String(Item)); 15/2

```
   procedure Put                                                           16/2
      (Item : in Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Text_IO.Put (Bounded.To_String(Item)); 17/2

```
   procedure Put_Line                                                      18/2
      (File : in File_Type;
       Item : in Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Text_IO.Put_Line (File, Bounded.To_String(Item)); 19/2

```
   procedure Put_Line                                                      20/2
      (Item : in Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Text_IO.Put_Line (Bounded.To_String(Item)); 21/2

```
   function Get_Line                                                       22/2
      (File : in File_Type)
      return Bounded.Bounded_String;
```

{*AI95-00428-01*} Returns Bounded.To_Bounded_String(Text_IO.Get_Line(File)); 23/2

```
   function Get_Line                                                       24/2
      return Bounded.Bounded_String;
```

{*AI95-00428-01*} Returns Bounded.To_Bounded_String(Text_IO.Get_Line); 25/2

```
   procedure Get_Line                                                      26/2
      (File : in File_Type; Item : out Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Item := Get_Line (File); 27/2

```
   procedure Get_Line                                                      28/2
      (Item : out Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Item := Get_Line; 29/2

Extensions to Ada 95

{*AI95-00428-01*} {*extensions to Ada 95*} Package Text_IO.Bounded_IO is new. 29.a/2

A.10.12 Input-Output for Unbounded Strings

{*AI95-00301-01*} The package Text_IO.Unbounded_IO provides input-output in human-readable form for Unbounded_Strings. 1/2

Static Semantics

2/2 {*AI95-00301-01*} The library package Text_IO.Unbounded_IO has the following declaration:

3/2
```
with Ada.Strings.Unbounded;
package Ada.Text_IO.Unbounded_IO is
```

4/2
```
   procedure Put
      (File : in File_Type;
       Item : in Strings.Unbounded.Unbounded_String);
```

5/2
```
   procedure Put
      (Item : in Strings.Unbounded.Unbounded_String);
```

6/2
```
   procedure Put_Line
      (File : in File_Type;
       Item : in Strings.Unbounded.Unbounded_String);
```

7/2
```
   procedure Put_Line
      (Item : in Strings.Unbounded.Unbounded_String);
```

8/2
```
   function Get_Line
      (File : in File_Type)
      return Strings.Unbounded.Unbounded_String;
```

9/2
```
   function Get_Line
      return Strings.Unbounded.Unbounded_String;
```

10/2
```
   procedure Get_Line
      (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);
```

11/2
```
   procedure Get_Line
      (Item : out Strings.Unbounded.Unbounded_String);
```

12/2
```
end Ada.Text_IO.Unbounded_IO;
```

13/2 {*AI95-00301-01*} For an item of type Unbounded_String, the following subprograms are provided:

14/2
```
procedure Put
   (File : in File_Type;
    Item : in Strings.Unbounded.Unbounded_String);
```

15/2 {*AI95-00301-01*} Equivalent to Text_IO.Put (File, Strings.Unbounded.To_String(Item));

16/2
```
procedure Put
   (Item : in Strings.Unbounded.Unbounded_String);
```

17/2 {*AI95-00301-01*} Equivalent to Text_IO.Put (Strings.Unbounded.To_String(Item));

18/2
```
procedure Put_Line
   (File : in File_Type;
    Item : in Strings.Unbounded.Unbounded_String);
```

19/2 {*AI95-00301-01*} Equivalent to Text_IO.Put_Line (File, Strings.Unbounded.To_String(Item));

20/2
```
procedure Put_Line
   (Item : in Strings.Unbounded.Unbounded_String);
```

21/2 {*AI95-00301-01*} Equivalent to Text_IO.Put_Line (Strings.Unbounded.To_String(Item));

22/2
```
function Get_Line
   (File : in File_Type)
   return Strings.Unbounded.Unbounded_String;
```

23/2 {*AI95-00301-01*} Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line(File));

24/2
```
function Get_Line
   return Strings.Unbounded.Unbounded_String;
```

25/2 {*AI95-00301-01*} Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line);

26/2
```
procedure Get_Line
   (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);
```

27/2 {*AI95-00301-01*} Equivalent to Item := Get_Line (File);

28/2
```
procedure Get_Line
   (Item : out Strings.Unbounded.Unbounded_String);
```

29/2 {*AI95-00301-01*} Equivalent to Item := Get_Line;

Extensions to Ada 95

29.a/2 {*AI95-00301-01*} {*extensions to Ada 95*} Package Text_IO.Unbounded_IO is new.

A.11 Wide Text Input-Output and Wide Wide Text Input-Output

1/2 {*AI95-00285-01*} The packages Wide_Text_IO and Wide_Wide_Text_IO provide facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of wide characters (or wide wide characters) grouped into lines, and as a sequence of lines grouped into pages.

Static Semantics

2/2 {*AI95-00285-01*} {*AI95-00301-01*} The specification of package Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line subprogram, any occurrence of Character is replaced by Wide_Character, and any occurrence of String is replaced by Wide_String. Nongeneric equivalents of Wide_Text_IO.Integer_IO and Wide_Text_IO.-Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_Wide_Text_IO, Ada.Long_Integer_Wide_Text_IO, Ada.Float_Wide_Text_IO, Ada.Long_-Float_Wide_Text_IO.

3/2 {*AI95-00285-01*} {*AI95-00301-01*} The specification of package Wide_Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line subprogram, any occurrence of Character is replaced by Wide_Wide_Character, and any occurrence of String is replaced by Wide_Wide_String. Nongeneric equivalents of Wide_Wide_Text_IO.Integer_IO and Wide_Wide_Text_IO.Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_Wide_Wide_Text_IO, Ada.Long_Integer_Wide_Wide_Text_IO, Ada.Float_-Wide_Wide_Text_IO, Ada.Long_Float_Wide_Wide_Text_IO.

4/2 {*AI95-00285-01*} {*AI95-00428-01*} The specification of package Wide_Text_IO.Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Wide_Bounded_String, and any occurrence of package Bounded is replaced by Wide_Bounded. The specification of package Wide_Wide_Text_IO.Wide_Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Wide_Wide_-Bounded_String, and any occurrence of package Bounded is replaced by Wide_Wide_Bounded.

5/2 {*AI95-00285-01*} {*AI95-00301-01*} The specification of package Wide_Text_IO.Wide_Unbounded_IO is the same as that for Text_IO.Unbounded_IO, except that any occurrence of Unbounded_String is replaced by Wide_Unbounded_String, and any occurrence of package Unbounded is replaced by Wide_-Unbounded. The specification of package Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO is the same as that for Text_IO.Unbounded_IO, except that any occurrence of Unbounded_String is replaced by Wide_Wide_Unbounded_String, and any occurrence of package Unbounded is replaced by Wide_Wide_-Unbounded.

Extensions to Ada 83

5.a {*extensions to Ada 83*} Support for Wide_Character and Wide_String I/O is new in Ada 95.

Extensions to Ada 95

5.b/2 {*AI95-00285-01*} {*extensions to Ada 95*} Package Wide_Wide_Text_IO is new. Be glad it wasn't called Double_Wide_Text_IO (for use in trailer parks) or Really_Wide_Text_IO.

5.c/2 {*AI95-00301-01*} Packages Wide_Text_IO.Wide_Unbounded_IO and Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO are also new.

5.d/2 {*AI95-00428-01*} Packages Wide_Text_IO.Wide_Bounded_IO and Wide_Wide_Text_IO.Wide_Wide_Bounded_IO are new as well.

A.12 Stream Input-Output

1/2 {*AI95-00285-01*} The packages Streams.Stream_IO, Text_IO.Text_Streams, Wide_Text_IO.Text_Streams, and Wide_Wide_Text_IO.Text_Streams provide stream-oriented operations on files.

Wording Changes from Ada 95

1.a/2 {*AI95-00285-01*} Included package Wide_Wide_Text_IO.Text_Streams in this description.

A.12.1 The Package Streams.Stream_IO

1 {*heterogeneous input-output*} [The subprograms in the child package Streams.Stream_IO provide control over stream files. Access to a stream file is either sequential, via a call on Read or Write to transfer an array of stream elements, or positional (if supported by the implementation for the given file), by specifying a relative index for an element. Since a stream file can be converted to a Stream_Access value, calling stream-oriented attribute subprograms of different element types with the same Stream_Access value provides heterogeneous input-output.] See 13.13 for a general discussion of streams.

Static Semantics

1.1/1 {*8652/0055*} {*AI95-00026-01*} The elements of a stream file are stream elements. If positioning is supported for the specified external file, a current index and current size are maintained for the file as described in A.8. If positioning is not supported, a current index is not maintained, and the current size is implementation defined.{*Current index (of an open stream file)*} {*Current size (of a stream file)*}

1.a.1/1 **Implementation defined:** Current size for a stream file for which positioning is not supported.

2 The library package Streams.Stream_IO has the following declaration:

```
3   with Ada.IO_Exceptions;
    package Ada.Streams.Stream_IO is

4       type Stream_Access is access all Root_Stream_Type'Class;

5       type File_Type is limited private;

6       type File_Mode is (In_File, Out_File, Append_File);

7       type    Count          is range 0 .. implementation-defined;
        subtype Positive_Count is Count range 1 .. Count'Last;
          -- Index into file, in stream elements.

8       procedure Create (File : in out File_Type;
                          Mode : in File_Mode := Out_File;
                          Name : in String    := "";
                          Form : in String    := "");

9       procedure Open (File : in out File_Type;
                        Mode : in File_Mode;
                        Name : in String;
                        Form : in String := "");

10      procedure Close  (File : in out File_Type);
        procedure Delete (File : in out File_Type);
        procedure Reset  (File : in out File_Type; Mode : in File_Mode);
        procedure Reset  (File : in out File_Type);
```

11
```
    function Mode (File : in File_Type) return File_Mode;
    function Name (File : in File_Type) return String;
    function Form (File : in File_Type) return String;
```
12
```
    function Is_Open     (File : in File_Type) return Boolean;
    function End_Of_File (File : in File_Type) return Boolean;
```
13
```
    function Stream (File : in File_Type) return Stream_Access;
        -- Return stream access for use with T'Input and T'Output
```
14/1 *This paragraph was deleted.*

15
```
    -- Read array of stream elements from file
    procedure Read (File : in  File_Type;
                    Item : out Stream_Element_Array;
                    Last : out Stream_Element_Offset;
                    From : in  Positive_Count);
```
16
```
    procedure Read (File : in  File_Type;
                    Item : out Stream_Element_Array;
                    Last : out Stream_Element_Offset);
```
17/1 *This paragraph was deleted.*

18
```
    -- Write array of stream elements into file
    procedure Write (File : in File_Type;
                     Item : in Stream_Element_Array;
                     To   : in Positive_Count);
```
19
```
    procedure Write (File : in File_Type;
                           Item : in Stream_Element_Array);
```
20/1 *This paragraph was deleted.*

21
```
    -- Operations on position within file
```
22
```
    procedure Set_Index(File : in File_Type; To : in Positive_Count);
```
23
```
    function Index(File : in File_Type) return Positive_Count;
    function Size (File : in File_Type) return Count;
```
24
```
    procedure Set_Mode(File : in out File_Type; Mode : in File_Mode);
```
25/1 {*8652/0051*} {*AI95-00057-01*} **procedure** Flush(File : **in** File_Type);

26
```
    -- exceptions
    Status_Error : exception renames IO_Exceptions.Status_Error;
    Mode_Error   : exception renames IO_Exceptions.Mode_Error;
    Name_Error   : exception renames IO_Exceptions.Name_Error;
    Use_Error    : exception renames IO_Exceptions.Use_Error;
    Device_Error : exception renames IO_Exceptions.Device_Error;
    End_Error    : exception renames IO_Exceptions.End_Error;
    Data_Error   : exception renames IO_Exceptions.Data_Error;
```
27
```
private
   ... -- not specified by the language
end Ada.Streams.Stream_IO;
```

27.1/2 {*AI95-00360-01*} The type File_Type needs finalization (see 7.6).

28/2 {*AI95-00283-01*} The subprograms given in subclause A.8.2 for the control of external files (Create, Open, Close, Delete, Reset, Mode, Name, Form, and Is_Open) are available for stream files.

28.1/2 {*AI95-00283-01*} The End_Of_File function:

- 28.2/2 Propagates Mode_Error if the mode of the file is not In_File;
- 28.3/2 If positioning is supported for the given external file, the function returns True if the current index exceeds the size of the external file; otherwise it returns False;
- 28.4/2 If positioning is not supported for the given external file, the function returns True if no more elements can be read from the given file; otherwise it returns False.

28.5/2 {*8652/0055*} {*AI95-00026-01*} {*AI95-00085-01*} The Set_Mode procedure sets the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

28.6/1 {*8652/0055*} {*AI95-00026-01*} The Flush procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file or changing the position. Mode_Error is propagated if the mode of the file is In_File.

29/1 {*8652/0056*} {*AI95-00001-01*} The Stream function returns a Stream_Access result from a File_Type object, thus allowing the stream-oriented attributes Read, Write, Input, and Output to be used on the same file for multiple types. Stream propagates Status_Error if File is not open.

30/2 {*AI95-00256-01*} The procedures Read and Write are equivalent to the corresponding operations in the package Streams. Read propagates Mode_Error if the mode of File is not In_File. Write propagates Mode_Error if the mode of File is not Out_File or Append_File. The Read procedure with a Positive_Count parameter starts reading at the specified index. The Write procedure with a Positive_Count parameter starts writing at the specified index. For a file that supports positioning, Read without a Positive_Count parameter starts reading at the current index, and Write without a Positive_Count parameter starts writing at the current index.

30.1/1 {*8652/0055*} {*AI95-00026-01*} The Size function returns the current size of the file.

31/1 {*8652/0055*} {*AI95-00026-01*} The Index function returns the current index.

31.a/1 *This paragraph was deleted.*

32 The Set_Index procedure sets the current index to the specified value.

32.1/1 {*8652/0055*} {*AI95-00026-01*} If positioning is supported for the external file, the current index is maintained as follows:

32.2/1 • {*8652/0055*} {*AI95-00026-01*} For Open and Create, if the Mode parameter is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one.

32.3/1 • {*8652/0055*} {*AI95-00026-01*} For Reset, if the Mode parameter is Append_File, or no Mode parameter is given and the current mode is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one.

32.4/1 • {*8652/0055*} {*AI95-00026-01*} For Set_Mode, if the new mode is Append_File, the current index is set to current size plus one; otherwise, the current index is unchanged.

32.5/1 • {*8652/0055*} {*AI95-00026-01*} For Read and Write without a Positive_Count parameter, the current index is incremented by the number of stream elements read or written.

32.6/1 • {*8652/0055*} {*AI95-00026-01*} For Read and Write with a Positive_Count parameter, the value of the current index is set to the value of the Positive_Count parameter plus the number of stream elements read or written.

33 If positioning is not supported for the given file, then a call of Index or Set_Index propagates Use_Error. Similarly, a call of Read or Write with a Positive_Count parameter propagates Use_Error.

33.a/2 **Implementation Note:** {*AI95-00085-01*} It is permissible for an implementation to implement mode Append_File using the Unix append mode (the O_APPEND bit). Such an implementation does not support positioning when the mode is Append_File, and therefore the operations listed above must raise Use_Error. This is acceptable as there is no requirement that any particular file support positioning; therefore it is acceptable that a file support positioning when opened with mode Out_File, and the same file not support positioning when opened with mode Append_File. But it is not acceptable for a file to support positioning (by allowing the above operations), but to do something other than the defined semantics (that is, always write at the end, even when explicitly commanded to write somewhere else).

Paragraphs 34 through 36 were deleted.

Erroneous Execution

{*8652/0056*} {*AI95-00001-01*} {*erroneous execution (cause)* [partial]} If the File_Type object passed to the Stream function is later closed or finalized, and the stream-oriented attributes are subsequently called (explicitly or implicitly) on the Stream_Access value returned by Stream, execution is erroneous. This rule applies even if the File_Type object was opened again after it had been closed. 36.1/1

Reason: These rules are analogous to the rule for the result of the Current_Input, Current_Output, and Current_Error functions. These rules make it possible to represent a value of (some descendant of) Root_Stream_Type which represents a file as an access value, with a null value corresponding to a closed file. 36.a.1/1

Inconsistencies With Ada 95

{*AI95-00283-01*} {*inconsistencies with Ada 95*} **Amendment Correction:** The description of the subprograms for managing files was corrected so that they do not require truncation of the external file — a stream file is not a sequential file. An Ada 95 program that expects truncation of the stream file may not work under Ada 2005. Note that the Ada 95 standard was ambiguous on this point (the normative wording seemed to require truncation, but didn't explain where; the AARM notes seemed to expect behavior like Direct_IO), and implementations varied widely. Therefore, as a practical matter, code that depends on stream truncation may not work even in Ada 95; deleting the file before opening it provides truncation that works in both Ada 95 and Ada 2005. 36.a/2

Incompatibilities With Ada 95

{*AI95-00360-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** Stream_IO.File_Type is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and File_Type does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in File_Type, and thus would be illegal. 36.b/2

Wording Changes from Ada 95

{*8652/0051*} {*AI95-00057-01*} **Corrigendum:** Corrected the parameter mode of Flush; otherwise it could not be used on Standard_Output. 36.c/2

{*8652/0055*} {*AI95-00026-01*} {*AI95-00256-01*} **Corrigendum:** Added wording to describe the effects of the various operations on the current index. The Amendment adds an explanation of the use of current index for Read and Write. 36.d/2

{*8652/0056*} {*AI95-00001-01*} **Corrigendum:** Clarified that Stream can raise Status_Error, and clarified that using a Stream_Access whose file has been closed is erroneous. 36.e/2

{*AI95-00085-01*} Clarified that Set_Mode can be called with the current mode. 36.f/2

A.12.2 The Package Text_IO.Text_Streams

The package Text_IO.Text_Streams provides a function for treating a text file as a stream. 1

Static Semantics

The library package Text_IO.Text_Streams has the following declaration: 2

3
```
with Ada.Streams;
package Ada.Text_IO.Text_Streams is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;
```
4
```
   function Stream (File : in File_Type) return Stream_Access;
end Ada.Text_IO.Text_Streams;
```

The Stream function has the same effect as the corresponding function in Streams.Stream_IO. 5

NOTES

34 The ability to obtain a stream for a text file allows Current_Input, Current_Output, and Current_Error to be processed with the functionality of streams, including the mixing of text and binary input-output, and the mixing of binary input-output for different types. 6

35 Performing operations on the stream associated with a text file does not affect the column, line, or page counts. 7

A.12.3 The Package Wide_Text_IO.Text_Streams

1 The package Wide_Text_IO.Text_Streams provides a function for treating a wide text file as a stream.

Static Semantics

2 The library package Wide_Text_IO.Text_Streams has the following declaration:

3
```
with Ada.Streams;
package Ada.Wide_Text_IO.Text_Streams is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;
```

4
```
   function Stream (File : in File_Type) return Stream_Access;
end Ada.Wide_Text_IO.Text_Streams;
```

5 The Stream function has the same effect as the corresponding function in Streams.Stream_IO.

A.12.4 The Package Wide_Wide_Text_IO.Text_Streams

1/2 {*AI95-00285-01*} The package Wide_Wide_Text_IO.Text_Streams provides a function for treating a wide wide text file as a stream.

Static Semantics

2/2 {*AI95-00285-01*} The library package Wide_Wide_Text_IO.Text_Streams has the following declaration:

3/2
```
with Ada.Streams;
package Ada.Wide_Wide_Text_IO.Text_Streams is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;
```

4/2
```
   function Stream (File : in File_Type) return Stream_Access;
end Ada.Wide_Wide_Text_IO.Text_Streams;
```

5/2 {*AI95-00285-01*} The Stream function has the same effect as the corresponding function in Streams.Stream_IO.

Extensions to Ada 95

5.a/2 {*AI95-00285-01*} {*extensions to Ada 95*} Package Wide_Wide_Text_IO.Text_Streams is new.

A.13 Exceptions in Input-Output

1 The package IO_Exceptions defines the exceptions needed by the predefined input-output packages.

Static Semantics

2 The library package IO_Exceptions has the following declaration:

3
```
package Ada.IO_Exceptions is
   pragma Pure(IO_Exceptions);
```

4
```
   Status_Error : exception;
   Mode_Error   : exception;
   Name_Error   : exception;
   Use_Error    : exception;
   Device_Error : exception;
   End_Error    : exception;
   Data_Error   : exception;
   Layout_Error : exception;
```

5
```
end Ada.IO_Exceptions;
```

6 If more than one error condition exists, the corresponding exception that appears earliest in the following list is the one that is propagated.

The exception Status_Error is propagated by an attempt to operate upon a file that is not open, and by an attempt to open a file that is already open. 7

The exception Mode_Error is propagated by an attempt to read from, or test for the end of, a file whose current mode is Out_File or Append_File, and also by an attempt to write to a file whose current mode is In_File. In the case of Text_IO, the exception Mode_Error is also propagated by specifying a file whose current mode is Out_File or Append_File in a call of Set_Input, Skip_Line, End_Of_Line, Skip_Page, or End_Of_Page; and by specifying a file whose current mode is In_File in a call of Set_Output, Set_Line_Length, Set_Page_Length, Line_Length, Page_Length, New_Line, or New_Page. 8

The exception Name_Error is propagated by a call of Create or Open if the string given for the parameter Name does not allow the identification of an external file. For example, this exception is propagated if the string is improper, or, alternatively, if either none or more than one external file corresponds to the string. 9

The exception Use_Error is propagated if an operation is attempted that is not possible for reasons that depend on characteristics of the external file. For example, this exception is propagated by the procedure Create, among other circumstances, if the given mode is Out_File but the form specifies an input only device, if the parameter Form specifies invalid access rights, or if an external file with the given name already exists and overwriting is not allowed. 10

The exception Device_Error is propagated if an input-output operation cannot be completed because of a malfunction of the underlying system. 11

The exception End_Error is propagated by an attempt to skip (read past) the end of a file. 12

The exception Data_Error can be propagated by the procedure Read (or by the Read attribute) if the element read cannot be interpreted as a value of the required subtype. This exception is also propagated by a procedure Get (defined in the package Text_IO) if the input character sequence fails to satisfy the required syntax, or if the value input does not belong to the range of the required subtype. 13

The exception Layout_Error is propagated (in text input-output) by Col, Line, or Page if the value returned exceeds Count'Last. The exception Layout_Error is also propagated on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, respectively (excluding the unbounded cases). It is also propagated by an attempt to Put too many characters to a string. 14

Documentation Requirements

The implementation shall document the conditions under which Name_Error, Use_Error and Device_Error are propagated. 15

Documentation Requirement: The conditions under which Io_Exceptions.Name_Error, Io_Exceptions.Use_Error, and Io_Exceptions.Device_Error are propagated. 15.a/2

Implementation Permissions

If the associated check is too complex, an implementation need not propagate Data_Error as part of a procedure Read (or the Read attribute) if the value read cannot be interpreted as a value of the required subtype. 16

Ramification: An example where the implementation may choose not to perform the check is an enumeration type with a representation clause with "holes" in the range of internal codes. 16.a

Erroneous Execution

{*erroneous execution (cause)* [partial]} [If the element read by the procedure Read (or by the Read attribute) cannot be interpreted as a value of the required subtype, but this is not detected and Data_Error is not propagated, then the resulting value can be abnormal, and subsequent references to the value can lead to 17 erroneous execution, as explained in 13.9.1. {*normal state of an object* [partial]} {*abnormal state of an object* [partial]}]

A.14 File Sharing

Dynamic Semantics

1 {*unspecified* [partial]} It is not specified by the language whether the same external file can be associated with more than one file object. If such sharing is supported by the implementation, the following effects are defined:

2 • Operations on one text file object do not affect the column, line, and page numbers of any other file object.

3/1 • *This paragraph was deleted.*{*8652/0057*} {*AI95-00050-01*}

4 • For direct and stream files, the current index is a property of each file object; an operation on one file object does not affect the current index of any other file object.

5 • For direct and stream files, the current size of the file is a property of the external file.

6 All other effects are identical.

Wording Changes from Ada 95

6.a/2 {*8652/0057*} {*AI95-00050-01*} **Corrigendum:** Removed the incorrect statement that the external files associated with the standard input, standard output, and standard error files are distinct.

A.15 The Package Command_Line

1 The package Command_Line allows a program to obtain the values of its arguments and to set the exit status code to be returned on normal termination.

1.a/2 **Implementation defined:** The meaning of Argument_Count, Argument, and Command_Name for package Command_Line. The bounds of type Command_Line.Exit_Status.

Static Semantics

2 The library package Ada.Command_Line has the following declaration:

```
3   package Ada.Command_Line is
      pragma Preelaborate(Command_Line);

4     function Argument_Count return Natural;

5     function Argument (Number : in Positive) return String;

6     function Command_Name return String;

7     type Exit_Status is implementation-defined integer type;

8     Success : constant Exit_Status;
      Failure : constant Exit_Status;

9     procedure Set_Exit_Status (Code : in Exit_Status);

10  private
      ... -- not specified by the language
    end Ada.Command_Line;
```

```
11  function Argument_Count return Natural;
```

12 If the external execution environment supports passing arguments to a program, then Argument_Count returns the number of arguments passed to the program invoking the function. Otherwise it returns 0. The meaning of “number of arguments” is implementation defined.

```
function Argument (Number : in Positive) return String;
```
13

If the external execution environment supports passing arguments to a program, then Argument returns an implementation-defined value corresponding to the argument at relative position Number. {*Constraint_Error (raised by failure of run-time check)*} If Number is outside the range 1..Argument_Count, then Constraint_Error is propagated. 14

Ramification: If the external execution environment does not support passing arguments to a program, then Argument(N) for any N will raise Constraint_Error, since Argument_Count is 0. 14.a

```
function Command_Name return String;
```
15

If the external execution environment supports passing arguments to a program, then Command_Name returns an implementation-defined value corresponding to the name of the command invoking the program; otherwise Command_Name returns the null string. 16

```
type Exit_Status is implementation-defined integer type;
```
16.1/1

The type Exit_Status represents the range of exit status values supported by the external execution environment. The constants Success and Failure correspond to success and failure, respectively. 17

```
procedure Set_Exit_Status (Code : in Exit_Status);
```
18

If the external execution environment supports returning an exit status from a program, then Set_Exit_Status sets Code as the status. Normal termination of a program returns as the exit status the value most recently set by Set_Exit_Status, or, if no such value has been set, then the value Success. If a program terminates abnormally, the status set by Set_Exit_Status is ignored, and an implementation-defined exit status value is set. 19

If the external execution environment does not support returning an exit value from a program, then Set_Exit_Status does nothing. 20

Implementation Permissions

An alternative declaration is allowed for package Command_Line if different functionality is appropriate for the external execution environment. 21

NOTES

36 Argument_Count, Argument, and Command_Name correspond to the C language's argc, argv[n] (for n>0) and argv[0], respectively. 22

To be honest: The correspondence of Argument_Count to argc is not direct — argc would be one more than Argument_Count, since the argc count includes the command name, whereas Argument_Count does not. 22.a

Extensions to Ada 83

{*extensions to Ada 83*} This clause is new in Ada 95. 22.b

A.16 The Package Directories

{*AI95-00248-01*} The package Directories provides operations for manipulating files and directories, and their names. 1/2

Discussion: The notes for this clause contain the expected interpretations of some of the operations on various target systems. “Unix” refers to the UNIX® operating system, and in most cases also covers Unix-like systems such as Linux and POSIX. “Windows®” refers to the Microsoft® Windows® 2000 operating system and usually also covers most other versions that use the Win32 API. 1.a/2

Static Semantics

2/2 {*AI95-00248-01*} The library package Directories has the following declaration:

```
3/2   with Ada.IO_Exceptions;
      with Ada.Calendar;
      package Ada.Directories is

4/2      -- Directory and file operations:

5/2      function Current_Directory return String;

6/2      procedure Set_Directory (Directory : in String);

7/2      procedure Create_Directory (New_Directory : in String;
                                     Form          : in String := "");

8/2      procedure Delete_Directory (Directory : in String);

9/2      procedure Create_Path (New_Directory : in String;
                                Form          : in String := "");

10/2     procedure Delete_Tree (Directory : in String);

11/2     procedure Delete_File (Name : in String);

12/2     procedure Rename (Old_Name, New_Name : in String);

13/2     procedure Copy_File (Source_Name,
                              Target_Name : in String;
                              Form        : in String := "");

14/2     -- File and directory name operations:

15/2     function Full_Name (Name : in String) return String;

16/2     function Simple_Name (Name : in String) return String;

17/2     function Containing_Directory (Name : in String) return String;

18/2     function Extension (Name : in String) return String;

19/2     function Base_Name (Name : in String) return String;

20/2     function Compose (Containing_Directory : in String := "";
                          Name                 : in String;
                          Extension            : in String := "") return String;

21/2     -- File and directory queries:

22/2     type File_Kind is (Directory, Ordinary_File, Special_File);

23/2     type File_Size is range 0 .. implementation-defined;

24/2     function Exists (Name : in String) return Boolean;

25/2     function Kind (Name : in String) return File_Kind;

26/2     function Size (Name : in String) return File_Size;

27/2     function Modification_Time (Name : in String) return Ada.Calendar.Time;

28/2     -- Directory searching:

29/2     type Directory_Entry_Type is limited private;

30/2     type Filter_Type is array (File_Kind) of Boolean;

31/2     type Search_Type is limited private;

32/2     procedure Start_Search (Search    : in out Search_Type;
                                 Directory : in String;
                                 Pattern   : in String;
                                 Filter    : in Filter_Type := (others => True));

33/2     procedure End_Search (Search : in out Search_Type);

34/2     function More_Entries (Search : in Search_Type) return Boolean;

35/2     procedure Get_Next_Entry (Search : in out Search_Type;
                                   Directory_Entry : out Directory_Entry_Type);
```

36/2
```
    procedure Search (
       Directory : in String;
       Pattern   : in String;
       Filter    : in Filter_Type := (others => True);
       Process   : not null access procedure (
           Directory_Entry : in Directory_Entry_Type));
```
37/2
```
    -- Operations on Directory Entries:
```
38/2
```
    function Simple_Name (Directory_Entry : in Directory_Entry_Type)
        return String;
```
39/2
```
    function Full_Name (Directory_Entry : in Directory_Entry_Type)
        return String;
```
40/2
```
    function Kind (Directory_Entry : in Directory_Entry_Type)
        return File_Kind;
```
41/2
```
    function Size (Directory_Entry : in Directory_Entry_Type)
        return File_Size;
```
42/2
```
    function Modification_Time (Directory_Entry : in Directory_Entry_Type)
        return Ada.Calendar.Time;
```
43/2
```
    Status_Error : exception renames Ada.IO_Exceptions.Status_Error;
    Name_Error   : exception renames Ada.IO_Exceptions.Name_Error;
    Use_Error    : exception renames Ada.IO_Exceptions.Use_Error;
    Device_Error : exception renames Ada.IO_Exceptions.Device_Error;
```
44/2
```
 private
     -- Not specified by the language.
 end Ada.Directories;
```

45/2 {*AI95-00248-01*} External files may be classified as directories, special files, or ordinary files. A *directory* is an external file that is a container for files on the target system. A *special file* is an external file that cannot be created or read by a predefined Ada input-output package. External files that are not special files or directories are called *ordinary files*. {*directory*} {*special file*} {*ordinary file*}

45.a/2 **Ramification:** A directory is an external file, although it may not have a name on some targets. A directory is not a special file, as it can be created and read by Directories.

45.b/2 **Discussion:** Devices and soft links are examples of special files on Windows® and Unix.

45.c/2 Even if an implementation provides a package to create and read soft links, such links are still special files.

46/2 {*AI95-00248-01*} A *file name* is a string identifying an external file. Similarly, a *directory name* is a string identifying a directory. The interpretation of file names and directory names is implementation-defined. {*directory name*} {*file name*}

46.a/2 **Implementation defined:** The interpretation of file names and directory names.

47/2 {*AI95-00248-01*} The *full name* of an external file is a full specification of the name of the file. If the external environment allows alternative specifications of the name (for example, abbreviations), the full name should not use such alternatives. A full name typically will include the names of all of the directories that contain the item. The *simple name* of an external file is the name of the item, not including any containing directory names. Unless otherwise specified, a file name or directory name parameter in a call to a predefined Ada input-output subprogram can be a full name, a simple name, or any other form of name supported by the implementation. {*full name (of a file)*} {*simple name (of a file)*}

47.a/2 **Discussion:** The full name on Unix is a complete path to the root. For Windows®, the full name includes a complete path, as well as a disk name ("C:") or network share name. For both systems, the simple name is the part of the name following the last '/' (or '\' for Windows®). For example, in the name "/usr/randy/ada-directories.ads", "ada-directories.ads" is the simple name.

47.b/2 **Ramification:** It is possible for a file or directory name to be neither a full name nor a simple name. For instance, the Unix name "../parent/myfile" is neither a full name nor a simple name.

48/2 {*AI95-00248-01*} The *default directory* is the directory that is used if a directory or file name is not a full name (that is, when the name does not fully identify all of the containing directories). {*default directory*}

48.a/2 **Discussion:** The default directory is the one maintained by the familiar “cd” command on Unix and Windows®. Note that Windows® maintains separate default directories for each disk drive; implementations should use the natural implementation.

49/2 {*AI95-00248-01*} A *directory entry* is a single item in a directory, identifying a single external file (including directories and special files). {*directory entry*}

50/2 {*AI95-00248-01*} For each function that returns a string, the lower bound of the returned value is 1.

51/2 {*AI95-00248-01*} The following file and directory operations are provided:

52/2
```
function Current_Directory return String;
```

53/2 Returns the full directory name for the current default directory. The name returned shall be suitable for a future call to Set_Directory. The exception Use_Error is propagated if a default directory is not supported by the external environment.

54/2
```
procedure Set_Directory (Directory : in String);
```

55/2 Sets the current default directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support making Directory (in the absence of Name_Error) a default directory.

56/2
```
procedure Create_Directory (New_Directory : in String;
                            Form          : in String := "");
```

57/2 Creates a directory with name New_Directory. The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation-defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of a directory. The exception Use_Error is propagated if the external environment does not support the creation of a directory with the given name (in the absence of Name_Error) and form.

58/2
```
procedure Delete_Directory (Directory : in String);
```

59/2 Deletes an existing empty directory with name Directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory (or some portion of its contents) with the given name (in the absence of Name_Error).

60/2
```
procedure Create_Path (New_Directory : in String;
                       Form          : in String := "");
```

61/2 Creates zero or more directories with name New_Directory. Each non-existent directory named by New_Directory is created.[For example, on a typical Unix system, Create_Path ("/usr/me/my"); would create directory "me" in directory "usr", then create directory "my" in directory "me".] The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation-defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of any directory. The exception Use_Error is propagated if the external environment does not support the creation of any directories with the given name (in the absence of Name_Error) and form.

62/2
```
procedure Delete_Tree (Directory : in String);
```

63/2 Deletes an existing directory with name Directory. The directory and all of its contents (possibly including other directories) are deleted. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory or some portion of its contents with the given name (in the absence of Name_Error). If Use_Error is propagated, it is unspecified whether a portion of the contents of the directory is deleted.

```
procedure Delete_File (Name : in String);
```
64/2

65/2 Deletes an existing ordinary or special file with name Name. The exception Name_Error is propagated if the string given as Name does not identify an existing ordinary or special external file. The exception Use_Error is propagated if the external environment does not support the deletion of the file with the given name (in the absence of Name_Error).

```
procedure Rename (Old_Name, New_Name : in String);
```
66/2

67/2 Renames an existing external file (including directories) with name Old_Name to New_Name. The exception Name_Error is propagated if the string given as Old_Name does not identify an existing external file. The exception Use_Error is propagated if the external environment does not support the renaming of the file with the given name (in the absence of Name_Error). In particular, Use_Error is propagated if a file or directory already exists with name New_Name.

67.a/2 **Implementation Note:** This operation is expected to work within a a single directory, and implementers are encouraged to support it across directories on a single device. Copying files from one device to another is discouraged (that's what Copy_File is for). However, there is no requirement to detect file copying by the target system. If the target system has an API that gives that for “free”, it can be used. For Windows®, for instance, MoveFile can be used to implement Rename.

```
procedure Copy_File (Source_Name,
                     Target_Name : in String;
                     Form        : in String);
```
68/2

69/2 Copies the contents of the existing external file with name Source_Name to an external file with name Target_Name. The resulting external file is a duplicate of the source external file. The Form parameter can be used to give system-dependent characteristics of the resulting external file; the interpretation of the Form parameter is implementation-defined. Exception Name_Error is propagated if the string given as Source_Name does not identify an existing external ordinary or special file, or if the string given as Target_Name does not allow the identification of an external file. The exception Use_Error is propagated if the external environment does not support creating the file with the name given by Target_Name and form given by Form, or copying of the file with the name given by Source_Name (in the absence of Name_Error).

69.a/2 **Ramification:** Name_Error is always raised if Source_Name identifies a directory. It is up to the implementation whether special files can be copied, or if Use_Error will be raised.

70/2 {*AI95-00248-01*} The following file and directory name operations are provided:

```
function Full_Name (Name : in String) return String;
```
71/2

72/2 Returns the full name corresponding to the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

72.a/2 **Discussion:** Full name means that no abbreviations are used in the returned name, and that it is a full specification of the name. Thus, for Unix and Windows®, the result should be a full path that does not contain any "." or ".." directories. Typically, the default directory is used to fill in any missing information.

```
function Simple_Name (Name : in String) return String;
```
73/2

74/2 Returns the simple name portion of the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

75/2
```
function Containing_Directory (Name : in String) return String;
```

76/2 Returns the name of the containing directory of the external file (including directories) identified by Name. (If more than one directory can contain Name, the directory name returned is implementation-defined.) The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if the external file does not have a containing directory.

76.a/2 **Discussion:** This is purely a string manipulation function. If Name is not given as a full name, the containing directory probably won't be one, either. For example, if Containing_Directory ("..\AARM\RM-A-8") is called on Windows®, the result should be "..\AARM". If there is no path at all on the name, the result should be "." (which represents the current directory). Use Full_Name on the result of Containing_Directory if the full name is needed.

77/2
```
function Extension (Name : in String) return String;
```

78/2 Returns the extension name corresponding to Name. The extension name is a portion of a simple name (not including any separator characters), typically used to identify the file class. If the external environment does not have extension names, then the null string is returned. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file.

78.a/2 **Discussion:** For Unix and Windows®, the extension is the portion of the simple name following the rightmost period. For example, in the simple name "RM-A-8.html", the extension is "html".

79/2
```
function Base_Name (Name : in String) return String;
```

80/2 Returns the base name corresponding to Name. The base name is the remainder of a simple name after removing any extension and extension separators. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

80.a/2 **Discussion:** For Unix and Windows®, the base name is the portion of the simple name preceding the rightmost period (except for the special directory names "." and "..", whose Base_Name is "." and ".."). For example, in the simple name "RM-A-8.html", the base name is "RM-A-8".

81/2
```
function Compose (Containing_Directory : in String := "";
                  Name                 : in String;
                  Extension            : in String := "") return String;
```

82/2 Returns the name of the external file with the specified Containing_Directory, Name, and Extension. If Extension is the null string, then Name is interpreted as a simple name; otherwise Name is interpreted as a base name. The exception Name_Error is propagated if the string given as Containing_Directory is not null and does not allow the identification of a directory, or if the string given as Extension is not null and is not a possible extension, or if the string given as Name is not a possible simple name (if Extension is null) or base name (if Extension is non-null).

82.a/2 **Ramification:** The above definition implies that if the Extension is null, for Unix and Windows® no '.' is added to Name.

82.b/2 **Discussion:** If Name is null, Name_Error should be raised, as nothing is not a possible simple name or base name.

82.c/2 Generally, Compose(Containing_Directory(F), Base_Name(F),Extension(F)) = F. However, this is not true on Unix or Windows® for file names that end with a '.'; Compose(Base_Name("Fooey."),Extension("Fooey.")) = "Fooey". This is not a problem for Windows®, as the names have the same meaning with or without the '.', but these are different names for Unix. Thus, care needs to be taken on Unix; if Extension is null, Base_Name should be avoided. (That's not usually a problem with file names generated by a program.)

83/2 {*AI95-00248-01*} The following file and directory queries and types are provided:

84/2
```
type File_Kind is (Directory, Ordinary_File, Special_File);
```

85/2 The type File_Kind represents the kind of file represented by an external file or directory.

```
type File_Size is range 0 .. implementation-defined;
```
86/2

The type File_Size represents the size of an external file. 87/2

Implementation defined: The maximum value for a file size in Directories. 87.a/2

```
function Exists (Name : in String) return Boolean;
```
88/2

Returns True if an external file represented by Name exists, and False otherwise. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files). 89/2

```
function Kind (Name : in String) return File_Kind;
```
90/2

Returns the kind of external file represented by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. 91/2

```
function Size (Name : in String) return File_Size;
```
92/2

Returns the size of the external file represented by Name. The size of an external file is the number of stream elements contained in the file. If the external file is not an ordinary file, the result is implementation-defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Constraint_Error is propagated if the file size is not a value of type File_Size. 93/2

Implementation defined: The result for Directories.Size for a directory or special file 93.a/2

Discussion: We allow raising Constraint_Error, so that an implementation for a system with 64-bit file sizes does not need to support full numerics on 64-bit integers just to implement this package. Of course, if 64-bit integers are available on such a system, they should be used when defining type File_Size. 93.b/2

```
function Modification_Time (Name : in String) return Ada.Calendar.Time;
```
94/2

Returns the time that the external file represented by Name was most recently modified. If the external file is not an ordinary file, the result is implementation-defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Use_Error is propagated if the external environment does not support reading the modification time of the file with the name given by Name (in the absence of Name_Error). 95/2

Implementation defined: The result for Directories.Modification_Time for a directory or special file. 95.a/2

{*AI95-00248-01*} The following directory searching operations and types are provided: 96/2

```
type Directory_Entry_Type is limited private;
```
97/2

The type Directory_Entry_Type represents a single item in a directory. These items can only be created by the Get_Next_Entry procedure in this package. Information about the item can be obtained from the functions declared in this package. A default-initialized object of this type is invalid; objects returned from Get_Next_Entry are valid. 98/2

```
type Filter_Type is array (File_Kind) of Boolean;
```
99/2

The type Filter_Type specifies which directory entries are provided from a search operation. If the Directory component is True, directory entries representing directories are provided. If the Ordinary_File component is True, directory entries representing ordinary files are provided. If the Special_File component is True, directory entries representing special files are provided. 100/2

```
type Search_Type is limited private;
```
101/2

The type Search_Type contains the state of a directory search. A default-initialized Search_Type object has no entries available (function More_Entries returns False). Type Search_Type needs finalization (see 7.6). 102/2

103/2
```
procedure Start_Search (Search    : in out Search_Type;
                        Directory : in String;
                        Pattern   : in String;
                        Filter    : in Filter_Type := (others => True));
```

104/2 Starts a search in the directory named by Directory for entries matching Pattern. Pattern represents a pattern for matching file names. If Pattern is null, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation-defined. Only items that match Filter will be returned. After a successful call on Start_Search, the object Search may have entries available, but it may have no entries available if no files or directories match Pattern and Filter. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error). When Start_Search propagates Name_Error or Use_Error, the object Search will have no entries available.

104.a/2 **Implementation defined:** The interpretation of a non-null search pattern in Directories.

105/2
```
procedure End_Search (Search : in out Search_Type);
```

106/2 Ends the search represented by Search. After a successful call on End_Search, the object Search will have no entries available.

106.a/2 **Ramification:** The only way that a call to End_Search could be unsuccessful if Device_Error (see A.13) is raised because of an underlying failure (or bug).

107/2
```
function More_Entries (Search : in Search_Type) return Boolean;
```

108/2 Returns True if more entries are available to be returned by a call to Get_Next_Entry for the specified search object, and False otherwise.

109/2
```
procedure Get_Next_Entry (Search : in out Search_Type;
                          Directory_Entry : out Directory_Entry_Type);
```

110/2 Returns the next Directory_Entry for the search described by Search that matches the pattern and filter. If no further matches are available, Status_Error is raised. It is implementation-defined as to whether the results returned by this routine are altered if the contents of the directory are altered while the Search object is valid (for example, by another program). The exception Use_Error is propagated if the external environment does not support continued searching of the directory represented by Search.

110.a/2 **Implementation defined:** The results of a Directories search if the contents of the directory are altered while a search is in progress.

111/2
```
procedure Search (
    Directory : in String;
    Pattern   : in String;
    Filter    : in Filter_Type := (others => True);
    Process   : not null access procedure (
        Directory_Entry : in Directory_Entry_Type));
```

112/2 Searches in the directory named by Directory for entries matching Pattern. The subprogram designated by Process is called with each matching entry in turn. Pattern represents a pattern for matching file names. If Pattern is null, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation-defined. Only items that match Filter will be returned. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error).

112.a/2 **Discussion:** "In turn" means that the calls to the subprogram designated by Process are not made in parallel; they can be made in any order but must be in sequence.

113/2
```
function Simple_Name (Directory_Entry : in Directory_Entry_Type)
     return String;
```

114/2 Returns the simple external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation-defined. The exception Status_Error is propagated if Directory_Entry is invalid.

115/2
```
function Full_Name (Directory_Entry : in Directory_Entry_Type)
     return String;
```

116/2 Returns the full external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation-defined. The exception Status_Error is propagated if Directory_Entry is invalid.

117/2
```
function Kind (Directory_Entry : in Directory_Entry_Type)
     return File_Kind;
```

118/2 Returns the kind of external file represented by Directory_Entry. The exception Status_Error is propagated if Directory_Entry is invalid.

119/2
```
function Size (Directory_Entry : in Directory_Entry_Type)
     return File_Size;
```

120/2 Returns the size of the external file represented by Directory_Entry. The size of an external file is the number of stream elements contained in the file. If the external file represented by Directory_Entry is not an ordinary file, the result is implementation-defined. The exception Status_Error is propagated if Directory_Entry is invalid. The exception Constraint_Error is propagated if the file size is not a value of type File_Size.

121/2
```
function Modification_Time (Directory_Entry : in Directory_Entry_Type)
     return Ada.Calendar.Time;
```

122/2 Returns the time that the external file represented by Directory_Entry was most recently modified. If the external file represented by Directory_Entry is not an ordinary file, the result is implementation-defined. The exception Status_Error is propagated if Directory_Entry is invalid. The exception Use_Error is propagated if the external environment does not support reading the modification time of the file represented by Directory_Entry.

Implementation Requirements

123/2 For Copy_File, if Source_Name identifies an existing external ordinary file created by a predefined Ada input-output package, and Target_Name and Form can be used in the Create operation of that input-output package with mode Out_File without raising an exception, then Copy_File shall not propagate Use_Error.

123.a/2 **Discussion:** This means that Copy_File will copy any file that the Ada programmer could copy (by writing some possibly complicated Ada code).

Implementation Advice

124/2 If other information about a file (such as the owner or creation date) is available in a directory entry, the implementation should provide functions in a child package Directories.Information to retrieve it.

124.a/2 **Implementation Advice:** Package Directories.Information should be provided to retrieve other information about a file.

124.b/2 **Implementation Note:** For Windows®, Directories.Information should contain at least the following routines:

124.c/2
```
package Ada.Directories.Information is
    -- System-specific directory information.
    -- Version for the Microsoft® Windows® operating system.
```

```
124.d/2         function Creation_Time (Name : in String) return Ada.Calendar.Time;
124.e/2         function Last_Access_Time (Name : in String) return Ada.Calendar.Time;
124.f/2         function Is_Read_Only (Name : in String) return Boolean;
124.g/2         function Needs_Archiving (Name : in String) return Boolean;
                    -- This generally means that the file needs to be backed up.
                    -- The flag is only cleared by backup programs.
124.h/2         function Is_Compressed (Name : in String) return Boolean;
124.i/2         function Is_Encrypted (Name : in String) return Boolean;
124.j/2         function Is_Hidden (Name : in String) return Boolean;
124.k/2         function Is_System (Name : in String) return Boolean;
124.l/2         function Is_Offline (Name : in String) return Boolean;
124.m/2         function Is_Temporary (Name : in String) return Boolean;
124.n/2         function Is_Sparse (Name : in String) return Boolean;
124.o/2         function Is_Not_Indexed (Name : in String) return Boolean;
124.p/2         function Creation_Time (Directory_Entry : in Directory_Entry_Type)
                     return Ada.Calendar.Time;
124.q/2         function Last_Access_Time (Directory_Entry : in Directory_Entry_Type)
                     return Ada.Calendar.Time;
124.r/2         function Is_Read_Only (Directory_Entry : in Directory_Entry_Type) return Boolean;
124.s/2         function Needs_Archiving (Directory_Entry : in Directory_Entry_Type) return
            Boolean;
                    -- This generally means that the file needs to be backed up.
                    -- The flag is only cleared by backup programs.
124.t/2         function Is_Compressed (Directory_Entry : in Directory_Entry_Type) return Boolean;
124.u/2         function Is_Encrypted (Directory_Entry : in Directory_Entry_Type) return Boolean;
124.v/2         function Is_Hidden (Directory_Entry : in Directory_Entry_Type) return Boolean;
124.w/2         function Is_System (Directory_Entry : in Directory_Entry_Type) return Boolean;
124.x/2         function Is_Offline (Directory_Entry : in Directory_Entry_Type) return Boolean;
124.y/2         function Is_Temporary (Directory_Entry : in Directory_Entry_Type) return Boolean;
124.z/2         function Is_Sparse (Directory_Entry : in Directory_Entry_Type) return Boolean;
124.aa/2        function Is_Not_Indexed (Directory_Entry : in Directory_Entry_Type) return
            Boolean;
124.bb/2        -- Additional implementation-defined subprograms allowed here.
            end Ada.Directories.Information;
```

124.cc/2 For Unix-like systems (Unix, POSIX, Linux, etc.), Directories.Information should contain at least the following routines:

```
124.dd/2    package Ada.Directories.Information is
                -- System-specific directory information.
                -- Unix and similar systems version.
124.ee/2        function Last_Access_Time (Name : in String) return Ada.Calendar.Time;
124.ff/2        function Last_Status_Change_Time (Name : in String) return Ada.Calendar.Time;
124.gg/2        type Permission is
                  (Others_Execute, Others_Write, Others_Read,
                   Group_Execute,  Group_Write,  Group_Read,
                   Owner_Execute,  Owner_Write,  Owner_Read,
                   Set_Group_ID,   Set_User_ID);
124.hh/2        type Permission_Set_Type is array (Permission) of Boolean;
124.ii/2        function Permission_Set (Name : in String) return Permission_Set_Type;
124.jj/2        function Owner (Name : in String) return String;
                    -- Returns the image of the User_Id. If a definition of User_Id
                    -- is available, an implementation-defined version of Owner
                    -- returning User_Id should also be defined.
124.kk/2        function Group (Name : in String) return String;
                    -- Returns the image of the User_Id. If a definition of Group_Id
                    -- is available, an implementation-defined version of Group
                    -- returning Group_Id should also be defined.
124.ll/2        function Is_Block_Special_File (Name : in String) return Boolean;
124.mm/2        function Is_Character_Special_File (Name : in String) return Boolean;
```

```
    function Is_FIFO (Name : in String) return Boolean;                    124.nn/2
    function Is_Symbolic_Link (Name : in String) return Boolean;           124.oo/2
    function Is_Socket (Name : in String) return Boolean;                  124.pp/2
    function Last_Access_Time (Directory_Entry : in Directory_Entry_Type)  124.qq/2
       return Ada.Calendar.Time;
    function Last_Status_Change_Time (Directory_Entry : in Directory_Entry_Type)  124.rr/2
       return Ada.Calendar.Time;
    function Permission_Set (Directory_Entry : in Directory_Entry_Type)    124.ss/2
       return Permission_Set_Type;
    function Owner (Directory_Entry : in Directory_Entry_Type) return String;  124.tt/2
       -- See Owner above.
    function Group (Directory_Entry : in Directory_Entry_Type) return String;  124.uu/2
       -- See Group above.
    function Is_Block_Special_File (Directory_Entry : in Directory_Entry_Type)  124.vv/2
       return Boolean;
    function Is_Character_Special_File (Directory_Entry : in Directory_Entry_Type)  124.ww/2
       return Boolean;
    function Is_FIFO (Directory_Entry : in Directory_Entry_Type) return Boolean;  124.xx/2
    function Is_Symbolic_Link (Directory_Entry : in Directory_Entry_Type)  124.yy/2
       return Boolean;
    function Is_Socket (Directory_Entry : in Directory_Entry_Type) return Boolean;  124.zz/2
    -- Additional implementation-defined subprograms allowed here.         124.aaa/2
end Ada.Directories.Information;
```

We give these definitions to give guidance so that every implementation for a given target is not unnecessarily different. Implementers are encouraged to make packages for other targets as similar to these as possible. 124.bbb/2

Start_Search and Search should raise Use_Error if Pattern is malformed, but not if it could represent a file in the directory but does not actually do so. 125/2

Implementation Advice: Directories.Start_Search and Directories.Search should raise Use_Error for malformed patterns. 125.a/2

Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file. 126/2

Implementation Advice: Directories.Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file. 126.a/2

Discussion: “Supported” includes raising an exception if either name is malformed, the file to rename doesn't exist, insufficient permission for the operation exists, or similar problems. But this advice requires implementations to document what they do, and tells implementers that simply raising Use_Error isn't acceptable. 126.b/2

NOTES

37 The operations Containing_Directory, Full_Name, Simple_Name, Base_Name, Extension, and Compose operate on file names, not external files. The files identified by these operations do not need to exist. Name_Error is raised only if the file name is malformed and cannot possibly identify a file. Of these operations, only the result of Full_Name depends on the current default directory; the result of the others depends only on their parameters. 127/2

38 Using access types, values of Search_Type and Directory_Entry_Type can be saved and queried later. However, another task or application can modify or delete the file represented by a Directory_Entry_Type value or the directory represented by a Search_Type value; such a value can only give the information valid at the time it is created. Therefore, long-term storage of these values is not recommended. 128/2

39 If the target system does not support directories inside of directories, then Kind will never return Directory and Containing_Directory will always raise Use_Error. 129/2

40 If the target system does not support creation or deletion of directories, then Create_Directory, Create_Path, Delete_Directory, and Delete_Tree will always propagate Use_Error. 130/2

41 To move a file or directory to a different location, use Rename. Most target systems will allow renaming of files from one directory to another. If the target file or directory might already exist, it should be deleted first. 131/2

Discussion: While Rename is only guaranteed to work for name changes within a single directory, its unlikely that implementers would purposely prevent functionality present in the underlying system from working. To move a file totally portably, it's necessary to handle failure of the Rename and fall back to Copy_File and Delete: 131.a/2

131.b
```
begin
   Rename (Source, Target);
exception
   when Use_Error =>
      Copy_File (Source, Target);
      Delete (Source);
end;
```

Extensions to Ada 95

131.c/2 {*AI95-00248-01*} {*extensions to Ada 95*} Package Ada.Directories is new.

A.17 The Package Environment_Variables

1/2 {*AI95-00370-01*} {*environment variable*} The package Environment_Variables allows a program to read or modify environment variables. Environment variables are name-value pairs, where both the name and value are strings. The definition of what constitutes an *environment variable*, and the meaning of the name and value, are implementation defined.

1.a/2 **Implementation defined:** The definition and meaning of an environment variable.

Static Semantics

2/2 {*AI95-00370-01*} The library package Environment_Variables has the following declaration:

3/2
```
package Ada.Environment_Variables is
   pragma Preelaborate(Environment_Variables);
```
4/2
```
   function Value (Name : in String) return String;
```
5/2
```
   function Exists (Name : in String) return Boolean;
```
6/2
```
   procedure Set (Name : in String; Value : in String);
```
7/2
```
   procedure Clear (Name : in String);
   procedure Clear;
```
8/2
```
   procedure Iterate (
       Process : not null access procedure (Name, Value : in String));
```
9/2
```
end Ada.Environment_Variables;
```
10/2
```
function Value (Name : in String) return String;
```

11/2 {*AI95-00370-01*} If the external execution environment supports environment variables, then Value returns the value of the environment variable with the given name. If no environment variable with the given name exists, then Constraint_Error is propagated. If the execution environment does not support environment variables, then Program_Error is propagated.

12/2
```
function Exists (Name : in String) return Boolean;
```

13/2 {*AI95-00370-01*} If the external execution environment supports environment variables and an environment variable with the given name currently exists, then Exists returns True; otherwise it returns False.

14/2
```
procedure Set (Name : in String; Value : in String);
```

15/2 {*AI95-00370-01*} If the external execution environment supports environment variables, then Set first clears any existing environment variable with the given name, and then defines a single new environment variable with the given name and value. Otherwise Program_Error is propagated.

16/2 If implementation-defined circumstances prohibit the definition of an environment variable with the given name and value, then Constraint_Error is propagated.

16.a/2 **Implementation defined:** The circumstances where an environment variable cannot be defined.

17/2 It is implementation defined whether there exist values for which the call Set(Name, Value) has the same effect as Clear (Name).

Implementation defined: Environment names for which Set has the effect of Clear. 17.a/2

```
procedure Clear (Name : in String);
```
18/2

{*AI95-00370-01*} If the external execution environment supports environment variables, then Clear deletes all existing environment variable with the given name. Otherwise Program_Error is propagated. 19/2

```
procedure Clear;
```
20/2

{*AI95-00370-01*} If the external execution environment supports environment variables, then Clear deletes all existing environment variables. Otherwise Program_Error is propagated. 21/2

```
procedure Iterate (
     Process : not null access procedure (Name, Value : in String));
```
22/2

{*AI95-00370-01*} If the external execution environment supports environment variables, then Iterate calls the subprogram designated by Process for each existing environment variable, passing the name and value of that environment variable. Otherwise Program_Error is propagated. 23/2

If several environment variables exist that have the same name, Process is called once for each such variable. 24/2

Bounded (Run-Time) Errors

{*AI95-00370-01*} {*bounded error (cause)* [partial]} It is a bounded error to call Value if more than one environment variable exists with the given name; the possible outcomes are that: 25/2

- one of the values is returned, and that same value is returned in subsequent calls in the absence of changes to the environment; or 26/2
- Program_Error is propagated. 27/2

Erroneous Execution

{*AI95-00370-01*} {*erroneous execution (cause)* [partial]} Making calls to the procedures Set or Clear concurrently with calls to any subprogram of package Environment_Variables, or to any instantiation of Iterate, results in erroneous execution. 28/2

Making calls to the procedures Set or Clear in the actual subprogram corresponding to the Process parameter of Iterate results in erroneous execution. 29/2

Documentation Requirements

{*AI95-00370-01*} An implementation shall document how the operations of this package behave if environment variables are changed by external mechanisms (for instance, calling operating system services). 30/2

Documentation Requirement: The behavior of package Environment_Variables when environment variables are changed by external mechanisms. 30.a/2

Implementation Permissions

{*AI95-00370-01*} An implementation running on a system that does not support environment variables is permitted to define the operations of package Environment_Variables with the semantics corresponding to the case where the external execution environment does support environment variables. In this case, it shall provide a mechanism to initialize a nonempty set of environment variables prior to the execution of a partition. 31/2

Implementation Advice

32/2 {*AI95-00370-01*} If the execution environment supports subprocesses, the currently defined environment variables should be used to initialize the environment variables of a subprocess.

32.a/2 **Implementation Advice:** If the execution environment supports subprocesses, the current environment variables should be used to initialize the environment variables of a subprocess.

33/2 Changes to the environment variables made outside the control of this package should be reflected immediately in the effect of the operations of this package. Changes to the environment variables made using this package should be reflected immediately in the external execution environment. This package should not perform any buffering of the environment variables.

33.a/2 **Implementation Advice:** Changes to the environment variables made outside the control of Environment_Variables should be reflected immediately.

Extensions to Ada 95

33.b/2 {*AI95-00370-01*} {*extensions to Ada 95*} Package Environment_Variables is new.

A.18 Containers

1/2 {*AI95-00302-03*} This clause presents the specifications of the package Containers and several child packages, which provide facilities for storing collections of elements.

2/2 {*AI95-00302-03*} A variety of sequence and associative containers are provided. Each container includes a *cursor* type. A cursor is a reference to an element within a container. Many operations on cursors are common to all of the containers. A cursor referencing an element in a container is considered to be overlapping with the container object itself.{*cursor (for a container)* [partial]} {*container (cursor)*}

2.a/2 **Reason:** The last sentence is intended to clarify that operations that just use a cursor are on the same footing as operations that use a container in terms of the reentrancy rules of Annex A.

3/2 {*AI95-00302-03*} Within this clause we provide Implementation Advice for the desired average or worst case time complexity of certain operations on a container. This advice is expressed using the Landau symbol *O*(X). Presuming f is some function of a length parameter N and t(N) is the time the operation takes (on average or worst case, as specified) for the length N, a complexity of *O*(f(N)) means that there exists a finite A such that for any N, t(N)/f(N) < A. {*Landau symbol O(X)*} {*O(f(N))*}

3.a/2 **Discussion:** Of course, an implementation can do better than a specified *O*(f(N)): for example, *O*(1) meets the requirements for *O*(log N).

3.b/2 This concept seems to have as many names as there are authors. We used “Landau symbol” because that's what our reference does. But we'd also seen this referred as big-O notation{*big-O notation*} (sometimes written as *big-oh*), and as Bachmann notation. Whatever the name, it always has the above definition.

4/2 If the advice suggests that the complexity should be less than *O*(f(N)), then for any arbitrarily small positive real D, there should exist a positive integer M such that for all N > M, t(N)/f(N) < D.

Language Design Principles

4.a/2 {*AI95-00302-03*} This clause provides a number of useful containers for Ada. Only the most useful containers are provided. Ones that are relatively easy to code, redundant, or rarely used are omitted from this set, even if they are generally included in containers libraries.

4.b/2 The containers packages are modeled on the Standard Template Library (STL), an algorithms and data structure library popularized by Alexander Stepanov, and included in the C++ standard library. The structure and terminology differ from the STL where that better maps to common Ada usage. For instance, what the STL calls “iterators” are called “cursors” here.

4.c/2 The following major nonlimited containers are provided:

4.d/2 • (Expandable) Vectors of any nonlimited type;

4.e/2 • Doubly-linked Lists of any nonlimited type;

- Hashed Maps keyed by any nonlimited hashable type, and containing any nonlimited type; 4.f/2
- Ordered Maps keyed by any nonlimited ordered type, and containing any nonlimited type; 4.g/2
- Hashed Sets of any nonlimited hashable type; and 4.h/2
- Ordered Sets of any nonlimited ordered type. 4.i/2

Separate versions for definite and indefinite element types are provided, as those for definite types can be implemented more efficiently. 4.j/2

Each container includes a cursor, which is a reference to an element within a container. Cursors generally remain valid as long as the container exists and the element referenced is not deleted. Many operations on cursors are common to all of the containers. This makes it possible to write generic algorithms that work on any kind of container. 4.k/2

The containers packages are structured so that additional packages can be added in the future. Indeed, we hope that these packages provide the basis for a more extensive secondary standard for containers. 4.l/2

If containers with similar functionality (but different performance characteristics) are provided (by the implementation or by a secondary standard), we suggest that a prefix be used to identify the class of the functionality: "Ada.Containers.Bounded_Sets" (for a set with a maximum number of elements); "Ada.Containers.Protected_Maps" (for a map which can be accessed by multiple tasks at one time); "Ada.Containers.Persistent_Vectors" (for a persistent vector which continues to exist between executions of a program) and so on. 4.m/2

Note that the language already includes several requirements that are important to the use of containers. These include: 4.n/2

- Library packages must be reentrant – multiple tasks can use the packages as long as they operate on separate containers. Thus, it is only necessary for a user to protect a container if a single container needs to be used by multiple tasks. 4.o/2
- Language-defined types must stream "properly". That means that the stream attributes can be used to implement persistence of containers when necessary, and containers can be passed between partitions of a program. 4.p/2
- Equality of language-defined types must compose “properly”. This means that the version of "=" directly used by users is the same one that will be used in generics and in predefined equality operators of types with components of the containers and/or cursors. This prevents the abstraction from breaking unexpectedly. 4.q/2

If a container's element type is controlled, the point at which the element is finalized will depend on the implementation of the container. We do not specify precisely where this will happen (it will happen no later than the finalization of the container, of course) in order to give implementation's flexibility to cache, block, or split the nodes of the container. In particular, Delete does not necessarily finalize the element; the implementation may (or may not) hold the space for reuse. 4.r/2

This is not likely to be a hardship, as the element type has to be nonlimited. Types used to manage scarce resources generally need to be limited. Otherwise, the amount of resources needed is hard to control, as the language allows a lot of variation in the number or order of adjusts/finalizations. For common uses of nonlimited controlled types such as managing storage, the types already have to manage arbitrary copies. 4.s/2

The use of controlled type also brings up the possibility of failure of finalization (and thus deallocation) of an element. This is a “serious bug”, as AI-179 puts it, so we don't try to specify what happens in that case. The implementation should propagate the exception. 4.t/2

Implementation Note: It is expected that exceptions propagated from these operations do not damage containers. That is, if Storage_Error is propagated because of an allocation failure, or Constraint_Error is propagated by the assignment of elements, the container can continue to be used without further exceptions. The intent is that it should be possible to recover from errors without losing data. We don't try to state this formally in most cases, because it is hard to define precisely what is and is not allowed behavior. 4.u/2

Implementation Note: When this clause says that the behavior of something is unspecified{*unspecified* [partial]} , we really mean that any result of executing Ada code short of erroneous execution is allowed. We do not mean that memory not belonging to the parameters of the operation can be trashed. When we mean to allow erroneous behavior, we specifically say that execution is erroneous. All this means if the containers are written in Ada is that checks should not be suppressed or removed assuming some behavior of other code, and that the implementation should take care to avoid creating internal dangling accesses by assuming behavior from generic formals that can't be guaranteed. We don't try to say this normatively because it would be fairly complex, and implementers are unlikely to increase their support costs by fielding implementations that are unstable if given buggy hash functions, et al. 4.v/2

Extensions to Ada 95

{*AI95-00302-03*} {*extensions to Ada 95*} This clause is new. It just provides an introduction to the following subclauses. 4.w/2

A.18.1 The Package Containers

1/2 {*AI95-00302-03*} The package Containers is the root of the containers subsystem.

Static Semantics

2/2 {*AI95-00302-03*} The library package Containers has the following declaration:

```
3/2    package Ada.Containers is
          pragma Pure(Containers);
4/2       type Hash_Type is mod implementation-defined;
5/2       type Count_Type is range 0 .. implementation-defined;
6/2    end Ada.Containers;
```

7/2 {*AI95-00302-03*} Hash_Type represents the range of the result of a hash function. Count_Type represents the (potential or actual) number of elements of a container.

7.a/2 **Implementation defined:** The value of Containers.Hash_Type'Modulus. The value of Containers.Count_Type'Last.

Implementation Advice

8/2 {*AI95-00302-03*} Hash_Type'Modulus should be at least 2**32. Count_Type'Last should be at least 2**31–1.

8.a/2 **Implementation Advice:** Containers.Hash_Type'Modulus should be at least 2**32. Containers.Count_Type'Last should be at least 2**31–1.

8.b/2 **Discussion:** This is not a requirement so that these types can be declared properly on machines with native sizes that are not 32 bits. For instance, a 24-bit target could use 2**24 for Hash_Type'Modulus.

Extensions to Ada 95

8.c/2 {*AI95-00302-03*} {*extensions to Ada 95*} The package Containers is new.

A.18.2 The Package Containers.Vectors

1/2 The language-defined generic package Containers.Vectors provides private types Vector and Cursor, and a set of operations for each type. A vector container allows insertion and deletion at any position, but it is specifically optimized for insertion and deletion at the high end (the end with the higher index) of the container. A vector container also provides random access to its elements.{*vector container*} {*container (vector)*}

2/2 {*length (of a vector container)* [partial]} {*capacity (of a vector)* [partial]} A vector container behaves conceptually as an array that expands as necessary as items are inserted. The *length* of a vector is the number of elements that the vector contains. The *capacity* of a vector is the maximum number of elements that can be inserted into the vector prior to it being automatically expanded.

3/2 Elements in a vector container can be referred to by an index value of a generic formal type. The first element of a vector always has its index value equal to the lower bound of the formal type.

4/2 {*empty element (of a vector)* [partial]} A vector container may contain *empty elements*. Empty elements do not have a specified value.

4.a/2 **Implementation Note:** Vectors are not intended to be sparse (that is, there are elements at all defined positions). Users are expected to use other containers (like a Map) when they need sparse structures (there is a Note to this effect at the end of this subclause).

4.b/2 The internal array is a conceptual model of a vector. There is no requirement for an implementation to be a single contiguous array.

Static Semantics

5/2 {*AI95-00302-03*} The generic library package Containers.Vectors has the following declaration:

6/2
```
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Vectors is
   pragma Preelaborate(Vectors);
```

7/2
```
   subtype Extended_Index is
      Index_Type'Base range
         Index_Type'First-1 ..
         Index_Type'Min (Index_Type'Base'Last - 1, Index_Type'Last) + 1;
   No_Index : constant Extended_Index := Extended_Index'First;
```

8/2
```
   type Vector is tagged private;
   pragma Preelaborable_Initialization(Vector);
```

9/2
```
   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);
```

10/2
```
   Empty_Vector : constant Vector;
```

11/2
```
   No_Element : constant Cursor;
```

12/2
```
   function "=" (Left, Right : Vector) return Boolean;
```

13/2
```
   function To_Vector (Length : Count_Type) return Vector;
```

14/2
```
   function To_Vector
     (New_Item : Element_Type;
      Length   : Count_Type) return Vector;
```

15/2
```
   function "&" (Left, Right : Vector) return Vector;
```

16/2
```
   function "&" (Left  : Vector;
                 Right : Element_Type) return Vector;
```

17/2
```
   function "&" (Left  : Element_Type;
                 Right : Vector) return Vector;
```

18/2
```
   function "&" (Left, Right  : Element_Type) return Vector;
```

19/2
```
   function Capacity (Container : Vector) return Count_Type;
```

20/2
```
   procedure Reserve_Capacity (Container : in out Vector;
                               Capacity  : in     Count_Type);
```

21/2
```
   function Length (Container : Vector) return Count_Type;
```

22/2
```
   procedure Set_Length (Container : in out Vector;
                         Length    : in     Count_Type);
```

23/2
```
   function Is_Empty (Container : Vector) return Boolean;
```

24/2
```
   procedure Clear (Container : in out Vector);
```

25/2
```
   function To_Cursor (Container : Vector;
                       Index     : Extended_Index) return Cursor;
```

26/2
```
   function To_Index (Position  : Cursor) return Extended_Index;
```

27/2
```
   function Element (Container : Vector;
                     Index     : Index_Type)
      return Element_Type;
```

28/2
```
   function Element (Position : Cursor) return Element_Type;
```

29/2
```
   procedure Replace_Element (Container : in out Vector;
                              Index     : in     Index_Type;
                              New_Item  : in     Element_Type);
```

30/2
```
   procedure Replace_Element (Container : in out Vector;
                              Position  : in     Cursor;
                              New_item  : in     Element_Type);
```

31/2
```
procedure Query_Element
  (Container : in Vector;
   Index     : in Index_Type;
   Process   : not null access procedure (Element : in Element_Type));
```
32/2
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```
33/2
```
procedure Update_Element
  (Container : in out Vector;
   Index     : in     Index_Type;
   Process   : not null access procedure
                   (Element : in out Element_Type));
```
34/2
```
procedure Update_Element
  (Container : in out Vector;
   Position  : in     Cursor;
   Process   : not null access procedure
                   (Element : in out Element_Type));
```
35/2
```
procedure Move (Target : in out Vector;
                Source : in out Vector);
```
36/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  New_Item  : in     Vector);
```
37/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Vector);
```
38/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Vector;
                  Position  :    out Cursor);
```
39/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```
40/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```
41/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```
42/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1);
```
43/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```
44/2
```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Vector);
```
45/2
```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1);
```
46/2
```
procedure Append (Container : in out Vector;
                  New_Item  : in     Vector);
```
47/2
```
procedure Append (Container : in out Vector;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

```
procedure Insert_Space (Container : in out Vector;                    48/2
                        Before    : in     Extended_Index;
                        Count     : in     Count_Type := 1);

procedure Insert_Space (Container : in out Vector;                    49/2
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);

procedure Delete (Container : in out Vector;                          50/2
                  Index     : in     Extended_Index;
                  Count     : in     Count_Type := 1);

procedure Delete (Container : in out Vector;                          51/2
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1);

procedure Delete_First (Container : in out Vector;                    52/2
                        Count     : in     Count_Type := 1);

procedure Delete_Last (Container : in out Vector;                     53/2
                       Count     : in     Count_Type := 1);

procedure Reverse_Elements (Container : in out Vector);               54/2

procedure Swap (Container : in out Vector;                            55/2
                I, J      : in     Index_Type);

procedure Swap (Container : in out Vector;                            56/2
                I, J      : in     Cursor);

function First_Index (Container : Vector) return Index_Type;          57/2

function First (Container : Vector) return Cursor;                    58/2

function First_Element (Container : Vector)                           59/2
   return Element_Type;

function Last_Index (Container : Vector) return Extended_Index;       60/2

function Last (Container : Vector) return Cursor;                     61/2

function Last_Element (Container : Vector)                            62/2
   return Element_Type;

function Next (Position : Cursor) return Cursor;                      63/2

procedure Next (Position : in out Cursor);                            64/2

function Previous (Position : Cursor) return Cursor;                  65/2

procedure Previous (Position : in out Cursor);                        66/2

function Find_Index (Container : Vector;                              67/2
                     Item      : Element_Type;
                     Index     : Index_Type := Index_Type'First)
   return Extended_Index;

function Find (Container : Vector;                                    68/2
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor;

function Reverse_Find_Index (Container : Vector;                      69/2
                             Item      : Element_Type;
                             Index     : Index_Type := Index_Type'Last)
   return Extended_Index;

function Reverse_Find (Container : Vector;                            70/2
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor;

function Contains (Container : Vector;                                71/2
                   Item      : Element_Type) return Boolean;

function Has_Element (Position : Cursor) return Boolean;              72/2
```

73/2
```
      procedure  Iterate
        (Container : in Vector;
         Process   : not null access procedure (Position : in Cursor));
```
74/2
```
      procedure Reverse_Iterate
        (Container : in Vector;
         Process   : not null access procedure (Position : in Cursor));
```
75/2
```
      generic
         with function "<" (Left, Right : Element_Type)
            return Boolean is <>;
      package Generic_Sorting is
```
76/2
```
         function Is_Sorted (Container : Vector) return Boolean;
```
77/2
```
         procedure Sort (Container : in out Vector);
```
78/2
```
         procedure Merge (Target  : in out Vector;
                          Source  : in out Vector);
```
79/2
```
      end Generic_Sorting;
```
80/2
```
   private
```
81/2
```
      ... -- not specified by the language
```
82/2
```
   end Ada.Containers.Vectors;
```

83/2 {*AI95-00302-03*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions defined to use it return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions defined to use it are unspecified.{*unspecified* [partial]}

83.a/2 **Ramification:** The “functions defined to use it” are Find, Find_Index, Reverse_Find, Reverse_Find_Index, and "=" for Vectors. This list is a bit too long to give explicitly.

83.b/2 If the actual function for "=" is not symmetric and consistent, the result returned by any of the functions defined to use "=" cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other “bad” behavior. And it can call "=" in whatever manner makes sense. But note that only the results of the functions defined to use "=" are unspecified; other subprograms are not allowed to break if "=" is bad.

84/2 {*AI95-00302-03*} The type Vector is used to represent vectors. The type Vector needs finalization (see 7.6).

85/2 {*AI95-00302-03*} Empty_Vector represents the empty vector object. It has a length of 0. If an object of type Vector is not otherwise initialized, it is initialized to the same value as Empty_Vector.

86/2 {*AI95-00302-03*} No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

87/2 {*AI95-00302-03*} The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

88/2 {*AI95-00302-03*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

88.a/2 **Reason:** A cursor will probably be implemented in terms of one or more access values, and the effects of streaming access values is unspecified. Rather than letting the user stream junk by accident, we mandate that streaming of cursors raise Program_Error by default. The attributes can always be specified if there is a need to support streaming.

89/2 {*AI95-00302-03*} No_Index represents a position that does not correspond to any element. The subtype Extended_Index includes the indices covered by Index_Type plus the value No_Index and, if it exists, the successor to the Index_Type'Last.

89.a/2 **Discussion:** We require the existence of Index_Type'First – 1, so that No_Index and Last_Index of an empty vector is well-defined. We don't require the existence of Index_Type'Last + 1, as it is only used as the position of insertions (and needs to be allowed only when inserting an empty vector).

{*AI95-00302-03*} [Some operations of this generic package have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced.] 90/2

{*AI95-00302-03*} {*tamper with cursors (of a vector)*} A subprogram is said to *tamper with cursors* of a vector object *V* if: 91/2

- it inserts or deletes elements of *V*, that is, it calls the Insert, Insert_Space, Clear, Delete, or Set_Length procedures with *V* as a parameter; or 92/2

  To be honest: Operations which are defined to be equivalent to a call on one of these operations also are included. Similarly, operations which call one of these as part of their definition are included. 92.a/2

- it finalizes *V*; or 93/2
- it calls the Move procedure with *V* as a parameter. 94/2

  Discussion: Swap, Sort, and Merge copy elements rather than reordering them, so they don't tamper with cursors. 94.a/2

{*AI95-00302-03*} {*tamper with elements (of a vector)*} A subprogram is said to *tamper with elements* of a vector object *V* if: 95/2

- it tampers with cursors of *V*; or 96/2
- it replaces one or more elements of *V*, that is, it calls the Replace_Element, Reverse_Elements, or Swap procedures or the Sort or Merge procedures of an instance of Generic_Sorting with *V* as a parameter. 97/2

  Reason: Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem, so Update_Element does not cause a problem. 97.a/2

98/2
```
function "=" (Left, Right : Vector) return Boolean;
```

{*AI95-00302-03*} If Left and Right denote the same vector object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise it returns True. Any exception raised during evaluation of element equality is propagated. 99/2

Implementation Note: This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality additional times once the answer has been determined. 99.a/2

100/2
```
function To_Vector (Length : Count_Type) return Vector;
```

{*AI95-00302-03*} Returns a vector with a length of Length, filled with empty elements. 101/2

102/2
```
function To_Vector
  (New_Item : Element_Type;
   Length   : Count_Type) return Vector;
```

{*AI95-00302-03*} Returns a vector with a length of Length, filled with elements initialized to the value New_Item. 103/2

104/2
```
function "&" (Left, Right : Vector) return Vector;
```

{*AI95-00302-03*} Returns a vector comprising the elements of Left followed by the elements of Right. 105/2

106/2
```
function "&" (Left  : Vector;
              Right : Element_Type) return Vector;
```

107/2 {*AI95-00302-03*} Returns a vector comprising the elements of Left followed by the element Right.

108/2
```
function "&" (Left  : Element_Type;
              Right : Vector) return Vector;
```

109/2 {*AI95-00302-03*} Returns a vector comprising the element Left followed by the elements of Right.

110/2
```
function "&" (Left, Right  : Element_Type) return Vector;
```

111/2 {*AI95-00302-03*} Returns a vector comprising the element Left followed by the element Right.

112/2
```
function Capacity (Container : Vector) return Count_Type;
```

113/2 {*AI95-00302-03*} Returns the capacity of Container.

114/2
```
procedure Reserve_Capacity (Container : in out Vector;
                            Capacity  : in     Count_Type);
```

115/2 {*AI95-00302-03*} Reserve_Capacity allocates new internal data structures such that the length of the resulting vector can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then copies the elements into the new data structures and deallocates the old data structures. Any exception raised during allocation is propagated and Container is not modified.

115.a/2 **Discussion:** Expanding the internal array can be done by allocating a new, longer array, copying the elements, and deallocating the original array. This may raise Storage_Error, or cause an exception from a controlled subprogram. We require that a failed Reserve_Capacity does not lose any elements if an exception occurs, but we do not require a specific order of evaluations or copying.

115.b/2 This routine is used to preallocate the internal array to the specified capacity such that future Inserts do not require memory allocation overhead. Therefore, the implementation should allocate the needed memory to make that true at this point, even though the visible semantics could be preserved by waiting until the memory is needed. This doesn't apply to the indefinite element container, because elements will have to be allocated individually.

115.c/2 The implementation does not have to contract the internal array if the capacity is reduced, as any capacity greater than or equal to the specified capacity is allowed.

116/2
```
function Length (Container : Vector) return Count_Type;
```

117/2 {*AI95-00302-03*} Returns the number of elements in Container.

118/2
```
procedure Set_Length (Container : in out Vector;
                      Length    : in     Count_Type);
```

119/2 {*AI95-00302-03*} If Length is larger than the capacity of Container, Set_Length calls Reserve_Capacity (Container, Length), then sets the length of the Container to Length. If Length is greater than the original length of Container, empty elements are added to Container; otherwise elements are removed from Container.

119.a/2 **Ramification:** No elements are moved by this operation; any new empty elements are added at the end. This follows from the rules that a cursor continues to designate the same element unless the routine is defined to make the cursor ambiguous or invalid; this operation does not do that.

120/2
```
function Is_Empty (Container : Vector) return Boolean;
```

121/2 {*AI95-00302-03*} Equivalent to Length (Container) = 0.

122/2
```
procedure Clear (Container : in out Vector);
```

123/2 {*AI95-00302-03*} Removes all the elements from Container. The capacity of Container does not change.

124/2
```
function To_Cursor (Container : Vector;
                    Index     : Extended_Index) return Cursor;
```

125/2 {*AI95-00302-03*} If Index is not in the range First_Index (Container) .. Last_Index (Container), then No_Element is returned. Otherwise, a cursor designating the element at position Index in Container is returned.

126/2
```
function To_Index (Position  : Cursor) return Extended_Index;
```

127/2 {*AI95-00302-03*} If Position is No_Element, No_Index is returned. Otherwise, the index (within its containing vector) of the element designated by Position is returned.

127.a/2 **Ramification:** This implies that the index is determinable from a bare cursor alone. The basic model is that a vector cursor is implemented as a record containing an access to the vector container and an index value. This does constrain implementations, but it also allows all of the cursor operations to be defined in terms of the corresponding index operation (which should be primary for a vector).

128/2
```
function Element (Container : Vector;
                  Index     : Index_Type)
   return Element_Type;
```

129/2 {*AI95-00302-03*} If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Element returns the element at position Index.

130/2
```
function Element (Position  : Cursor) return Element_Type;
```

131/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position.

132/2
```
procedure Replace_Element (Container : in out Vector;
                           Index     : in     Index_Type;
                           New_Item  : in     Element_Type);
```

133/2 {*AI95-00302-03*} If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise Replace_Element assigns the value New_Item to the element at position Index. Any exception raised during the assignment is propagated. The element at position Index is not an empty element after successful call to Replace_Element.

134/2
```
procedure Replace_Element (Container : in out Vector;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
```

135/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns New_Item to the element designated by Position. Any exception raised during the assignment is propagated. The element at Position is not an empty element after successful call to Replace_Element.

135.a/2 **Ramification:** Replace_Element and Update_Element are the only ways that an element can change from empty to non-empty. Also see the note following Update_Element.

136/2
```
procedure Query_Element
  (Container : in Vector;
   Index     : in Index_Type;
   Process   : not null access procedure (Element : in Element_Type));
```

137/2 {*AI95-00302-03*} If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element at position Index as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

137.a/2 **Reason:** The "tamper with the elements" check is intended to prevent the Element parameter of Process from being modified or deleted outside of Process. The check prevents data loss (if Element_Type is passed by copy) or erroneous execution (if Element_Type is an unconstrained type in an indefinite container).

138/2
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```

139/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

140/2
```
procedure Update_Element
  (Container : in out Vector;
   Index     : in     Index_Type;
   Process   : not null access procedure (Element : in out Element_Type));
```

141/2 {*AI95-00302-03*} If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Update_Element calls Process.**all** with the element at position Index as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

142/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

142.a/2 **Ramification:** This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place.

143/2 The element at position Index is not an empty element after successful completion of this operation.

143.a/2 **Ramification:** Since reading an empty element is a bounded error, attempting to use this procedure to replace empty elements may fail. Use Replace_Element to do that reliably.

144/2
```
procedure Update_Element
  (Container : in out Vector;
   Position  : in     Cursor;
   Process   : not null access procedure (Element : in out Element_Type));
```

145/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

146/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

147/2 The element designated by Position is not an empty element after successful completion of this operation.

148/2
```
procedure Move (Target : in out Vector;
                Source : in out Vector);
```

149/2 {*AI95-00302-03*} If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target); then, each element from Source is removed from Source and inserted into Target in the original order. The length of Source is 0 after a successful call to Move.

149.a/2 **Discussion:** The idea is that the internal array is removed from Source and moved to Target. (See the Implementation Advice for Move). If Capacity (Target) /= 0, the previous internal array may need to be deallocated. We don't mention this explicitly, because it is covered by the "no memory loss" Implementation Requirement.

150/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  New_Item  : in     Vector);
```

151/2 {*AI95-00302-03*} If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Length(New_Item) is 0, then Insert does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Length (New_Item); if the value of Last appropriate for length *NL* would be greater than Index_Type'Last then Constraint_Error is propagated.

152/2 If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert slides the elements in the range Before .. Last_Index (Container) up by Length(New_Item) positions, and then copies the elements of New_Item to the positions starting at Before. Any exception raised during the copying is propagated.

152.a/2 **Ramification:** Moving the elements does not necessarily involve copying. Similarly, since Reserve_Capacity does not require the copying of elements, it does not need to be explicitly called (the implementation can combine the operations if it wishes to).

153/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Vector);
```

154/2 {*AI95-00302-03*} If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, if Length(New_Item) is 0, then Insert does nothing. If Before is No_Element, then the call is equivalent to Insert (Container, Last_Index (Container) + 1, New_Item); otherwise the call is equivalent to Insert (Container, To_Index (Before), New_Item);

154.a/2 **Ramification:** The check on Before checks that the cursor does not belong to some other Container. This check implies that a reference to the container is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception).

155/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Vector;
                  Position  :    out Cursor);
```

156/2 {*AI95-00302-03*} If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert (Container, *T*, New_Item) is called, and then Position is set to To_Cursor (Container, *T*).

156.a/2 **Discussion:** The messy wording is needed because Before is invalidated by Insert, and we don't want Position to be invalid after this call. An implementation probably only needs to copy Before to Position.

157/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

158/2 {*AI95-00302-03*} Equivalent to Insert (Container, Before, To_Vector (New_Item, Count));

159/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

160/2 {*AI95-00302-03*} Equivalent to Insert (Container, Before, To_Vector (New_Item, Count));

161/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

162/2 {*AI95-00302-03*} Equivalent to Insert (Container, Before, To_Vector (New_Item, Count), Position);

163/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1);
```

164/2 {*AI95-00302-03*} If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, then Insert does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Count; if the value of Last appropriate for length *NL* would be greater than Index_Type'Last then Constraint_Error is propagated.

165/2 If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts elements that are initialized by default (see 3.3.1) in the positions starting at Before.

166/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

167/2 {*AI95-00302-03*} If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert (Container, *T*, Count) is called, and then Position is set to To_Cursor (Container, *T*).

167.a/2 **Reason:** This routine exists mainly to ease conversion between Vector and List containers. Unlike Insert_Space, this routine default initializes the elements it inserts, which can be more expensive for some element types.

168/2
```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Vector;
                   Count     : in     Count_Type := 1);
```

169/2 {*AI95-00302-03*} Equivalent to Insert (Container, First_Index (Container), New_Item).

170/2
```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1);
```

171/2 {*AI95-00302-03*} Equivalent to Insert (Container, First_Index (Container), New_Item, Count).

172/2
```
procedure Append (Container : in out Vector;
                  New_Item  : in     Vector);
```

173/2 {*AI95-00302-03*} Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item).

174/2
```
procedure Append (Container : in out Vector;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

175/2 {*AI95-00302-03*} Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item, Count).

176/2
```
procedure Insert_Space (Container : in out Vector;
                        Before    : in     Extended_Index;
                        Count     : in     Count_Type := 1);
```

177/2 {*AI95-00302-03*} If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, then Insert_Space does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Count; if the value of Last appropriate for length *NL* would be greater than Index_Type'Last then Constraint_Error is propagated.

178/2 If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert_Space slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts empty elements in the positions starting at Before.

179/2
```
procedure Insert_Space (Container : in out Vector;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);
```

180/2 {*AI95-00302-03*} If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert_Space (Container, *T*, Count) is called, and then Position is set to To_Cursor (Container, *T*).

181/2
```
procedure Delete (Container : in out Vector;
                  Index     : in     Extended_Index;
                  Count     : in     Count_Type := 1);
```

182/2 {*AI95-00302-03*} If Index is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, Delete has no effect. Otherwise Delete slides the elements (if any) starting at position Index + Count down to Index. Any exception raised during element assignment is propagated.

182.a/2 **Ramification:** If Index + Count >= Last_Index(Container), this effectively truncates the vector (setting Last_Index to Index – 1 and consequently sets Length to Index – Index_Type'First).

183/2
```
procedure Delete (Container : in out Vector;
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1);
```

184/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete (Container, To_Index (Position), Count) is called, and then Position is set to No_Element.

185/2
```
procedure Delete_First (Container : in out Vector;
                        Count     : in     Count_Type := 1);
```

186/2 {*AI95-00302-03*} Equivalent to Delete (Container, First_Index (Container), Count).

187/2
```
procedure Delete_Last (Container : in out Vector;
                       Count     : in     Count_Type := 1);
```

188/2 {*AI95-00302-03*} If Length (Container) <= Count then Delete_Last is equivalent to Clear (Container). Otherwise it is equivalent to Delete (Container, Index_Type'Val(Index_Type'Pos(Last_Index (Container)) – Count + 1), Count).

189/2
```
procedure Reverse_Elements (Container : in out List);
```

190/2 {*AI95-00302-03*} Reorders the elements of Container in reverse order.

190.a/2 **Discussion:** This can copy the elements of the vector — all cursors referencing the vector are ambiguous afterwards and may designate different elements afterwards.

191/2
```
procedure Swap (Container : in out Vector;
                I, J      : in     Index_Type);
```

192/2 {*AI95-00302-03*} If either I or J is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Swap exchanges the values of the elements at positions I and J.

192.a/2 **To be honest:** The implementation is not required to actually copy the elements if it can do the swap some other way. But it is allowed to copy the elements if needed.

193/2
```
procedure Swap (Container : in out Vector;
                I, J      : in     Cursor);
```

194/2 {*AI95-00302-03*} If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap exchanges the values of the elements designated by I and J.

194.a/2 **Ramification:** After a call to Swap, I designates the element value previously designated by J, and J designates the element value previously designated by I. The cursors do not become ambiguous from this operation.

194.b/2 **To be honest:** The implementation is not required to actually copy the elements if it can do the swap some other way. But it is allowed to copy the elements if needed.

195/2
```
function First_Index (Container : Vector) return Index_Type;
```

196/2 {*AI95-00302-03*} Returns the value Index_Type'First.

196.a/2 **Discussion:** We'd rather call this “First”, but then calling most routines in here with First (Some_Vect) would be ambiguous.

197/2
```
function First (Container : Vector) return Cursor;
```

198/2 {*AI95-00302-03*} If Container is empty, First returns No_Element. Otherwise, it returns a cursor that designates the first element in Container.

199/2
```
function First_Element (Container : Vector) return Element_Type;
```

200/2 {*AI95-00302-03*} Equivalent to Element (Container, First_Index (Container)).

201/2
```
function Last_Index (Container : Vector) return Extended_Index;
```

202/2 {*AI95-00302-03*} If Container is empty, Last_Index returns No_Index. Otherwise, it returns the position of the last element in Container.

203/2
```
function Last (Container : Vector) return Cursor;
```

204/2 {*AI95-00302-03*} If Container is empty, Last returns No_Element. Otherwise, it returns a cursor that designates the last element in Container.

205/2
```
function Last_Element (Container : Vector) return Element_Type;
```

206/2 {*AI95-00302-03*} Equivalent to Element (Container, Last_Index (Container)).

207/2
```
function Next (Position : Cursor) return Cursor;
```

208/2 {*AI95-00302-03*} If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the element with index To_Index (Position) + 1 in the same vector as Position.

209/2
```
procedure Next (Position : in out Cursor);
```

210/2 {*AI95-00302-03*} Equivalent to Position := Next (Position).

```
function Previous (Position : Cursor) return Cursor;
```
211/2

{*AI95-00302-03*} If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the element with index To_Index (Position) – 1 in the same vector as Position. 212/2

```
procedure Previous (Position : in out Cursor);
```
213/2

{*AI95-00302-03*} Equivalent to Position := Previous (Position). 214/2

```
function Find_Index (Container : Vector;
                     Item      : Element_Type;
                     Index     : Index_Type := Index_Type'First)
   return Extended_Index;
```
215/2

{*AI95-00302-03*} Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index and proceeds towards Last_Index (Container). If no equal element is found, then Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered. 216/2

```
function Find (Container : Vector;
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor;
```
217/2

{*AI95-00302-03*} If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the first element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the last element of Container. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 218/2

```
function Reverse_Find_Index (Container : Vector;
                             Item      : Element_Type;
                             Index     : Index_Type := Index_Type'Last)
   return Extended_Index;
```
219/2

{*AI95-00302-03*} Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index or, if Index is greater than Last_Index (Container), at position Last_Index (Container). It proceeds towards First_Index (Container). If no equal element is found, then Reverse_Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered. 220/2

```
function Reverse_Find (Container : Vector;
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor;
```
221/2

{*AI95-00302-03*} If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise Reverse_Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the last element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the first element of Container. If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 222/2

```
function Contains (Container : Vector;
                   Item      : Element_Type) return Boolean;
```
223/2

{*AI95-00302-03*} Equivalent to Has_Element (Find (Container, Item)). 224/2

225/2
```
function Has_Element (Position : Cursor) return Boolean;
```

226/2 {*AI95-00302-03*} Returns True if Position designates an element, and returns False otherwise.

226.a/2 **To be honest:** This function may not detect cursors that designate deleted elements; such cursors are invalid (see below) and the result of calling Has_Element with an invalid cursor is unspecified (but not erroneous).

227/2
```
procedure Iterate
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor));
```

228/2 {*AI95-00302-03*} Invokes Process.**all** with a cursor that designates each element in Container, in index order. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process is propagated.

228.a/2 **Discussion:** The purpose of the "tamper with the cursors" check is to prevent erroneous execution from the Position parameter of Process.**all** becoming invalid. This check takes place when the operations that tamper with the cursors of the container are called. The check cannot be made later (say in the body of Iterate), because that could cause the Position cursor to be invalid and potentially cause execution to become erroneous -- defeating the purpose of the check.

228.b/2 There is no check needed if an attempt is made to insert or delete nothing (that is, Count = 0 or Length(Item) = 0).

228.c/2 The check is easy to implement: each container needs a counter. The counter is incremented when Iterate is called, and decremented when Iterate completes. If the counter is nonzero when an operation that inserts or deletes is called, Finalize is called, or one of the other operations in the list occurs, Program_Error is raised.

229/2
```
procedure Reverse_Iterate
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor));
```

230/2 {*AI95-00302-03*} Iterates over the elements in Container as per Iterate, except that elements are traversed in reverse index order.

231/2 The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.{*unspecified* [partial]}

232/2
```
function Is_Sorted (Container : Vector) return Boolean;
```

233/2 {*AI95-00302-03*} Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

234/2
```
procedure Sort (Container : in out Vector);
```

235/2 {*AI95-00302-03*} Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

235.a/2 **Ramification:** This implies swapping the elements, usually including an intermediate copy. This means that the elements will usually be copied. (As with Swap, if the implementation can do this some other way, it is allowed to.) Since the elements are nonlimited, this usually will not be a problem. Note that there is Implementation Advice below that the implementation should use a sort that minimizes copying of elements.

235.b/2 The sort is not required to be stable (and the fast algorithm required will not be stable). If a stable sort is needed, the user can include the original location of the element as an extra "sort key". We considered requiring the implementation to do that, but it is mostly extra overhead -- usually there is something already in the element that provides the needed stability.

```
    procedure Merge (Target  : in out Vector;
                     Source  : in out Vector);
```
236/2

{*AI95-00302-03*} Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated. 237/2

Discussion: It is a bounded error if either of the vectors is unsorted, see below. The bounded error can be recovered by sorting Target after the merge call, or the vectors can be pretested with Is_Sorted. 237.a/2

Implementation Note: The Merge operation will usually require copying almost all of the elements. One implementation strategy would be to extend Target to the appropriate length, then copying elements from the back of the vectors working towards the front. An alternative approach would be to allocate a new internal data array of the appropriate length, copy the elements into it in an appropriate order, and then replacing the data array in Target with the temporary. 237.b/2

Bounded (Run-Time) Errors

{*AI95-00302-03*} {*bounded error (cause)* [partial]} Reading the value of an empty element by calling Element, Query_Element, Update_Element, Swap, Is_Sorted, Sort, Merge, "=", Find, or Reverse_Find is a bounded error. The implementation may treat the element as having any normal value (see 13.9.1) of the element type, or raise Constraint_Error or Program_Error before modifying the vector. 238/2

Ramification: For instance, a default initialized element could be returned. Or some previous value of an element. But returning random junk is not allowed if the type has default initial value(s). 238.a/2

Assignment and streaming of empty elements are **not** bounded errors. This is consistent with regular composite types, for which assignment and streaming of uninitialized components do not cause a bounded error, but reading the uninitialized component does cause a bounded error. 238.b/2

There are other operations which are defined in terms of the operations listed above. 238.c/2

{*AI95-00302-03*} {*bounded error (cause)* [partial]} Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined. 239/2

{*AI95-00302-03*} {*ambiguous cursor (of a vector)*} {*cursor (ambiguous)*} A Cursor value is *ambiguous* if any of the following have occurred since it was created: 240/2

- Insert, Insert_Space, or Delete has been called on the vector that contains the element the cursor designates with an index value (or a cursor designating an element at such an index value) less than or equal to the index value of the element designated by the cursor; or 241/2
- The vector that contains the element it designates has been passed to the Sort or Merge procedures of an instance of Generic_Sorting, or to the Reverse_Elements procedure. 242/2

{*AI95-00302-03*} {*bounded error (cause)* [partial]} It is a bounded error to call any subprogram other than "=" or Has_Element declared in Containers.Vectors with an ambiguous (but not invalid, see below) cursor parameter. Possible results are: 243/2

- The cursor may be treated as if it were No_Element; 244/2
- The cursor may designate some element in the vector (but not necessarily the element that it originally designated); 245/2
- Constraint_Error may be raised; or 246/2
- Program_Error may be raised. 247/2

Reason: Cursors are made ambiguous if an Insert or Delete occurs that moves the elements in the internal array including the designated ones. After such an operation, the cursor probably still designates an element (although it 247.a/2 might not after a deletion), but it is a *different* element. That violates the definition of cursor — it designates a particular element.

247.b/2 For "=" or Has_Element, the cursor works normally (it would not be No_Element). We don't want to trigger an exception simply for comparing a bad cursor.

247.c/2 While it is possible to check for these cases or ensure that cursors survive such operations, in many cases the overhead necessary to make the check (or ensure cursors continue to designate the same element) is substantial in time or space.

Erroneous Execution

248/2 {*AI95-00302-03*} A Cursor value is *invalid* if any of the following have occurred since it was created:{*invalid cursor (of a vector)*} {*cursor (invalid)* [partial]}

249/2 • The vector that contains the element it designates has been finalized;

250/2 • The vector that contains the element it designates has been used as the Source or Target of a call to Move; or

251/2 • The element it designates has been deleted.

252/2 {*AI95-00302-03*} The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter.{*unspecified* [partial]} Execution is erroneous if any other subprogram declared in Containers.Vectors is called with an invalid cursor parameter.{*erroneous execution (cause)* [partial]}

252.a/2 **Discussion:** The list above (combined with the bounded error cases) is intended to be exhaustive. In other cases, a cursor value continues to designate its original element. For instance, cursor values survive the appending of new elements.

Implementation Requirements

253/2 {*AI95-00302-03*} No storage associated with a vector object shall be lost upon assignment or scope exit.

254/2 {*AI95-00302-03*} The execution of an assignment_statement for a vector shall have the effect of copying the elements from the source vector object to the target vector object.

254.a/2 **Implementation Note:** An assignment of a Vector is a “deep” copy; that is the elements are copied as well as the data structures. We say “effect of” in order to allow the implementation to avoid copying elements immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed.

Implementation Advice

255/2 {*AI95-00302-03*} Containers.Vectors should be implemented similarly to an array. In particular, if the length of a vector is *N*, then

256/2 • the worst-case time complexity of Element should be *O*(log *N*);

256.a/2 **Implementation Advice:** The worst-case time complexity of Element for Containers.Vector should be *O*(log *N*).

257 • the worst-case time complexity of Append with Count=1 when *N* is less than the capacity of the vector should be *O*(log *N*); and

257.a/2 **Implementation Advice:** The worst-case time complexity of Append with Count = 1 when *N* is less than the capacity for Containers.Vector should be *O*(log *N*).

258/2 • the worst-case time complexity of Prepend with Count=1 and Delete_First with Count=1 should be *O*(*N* log *N*).

258.a/2 **Implementation Advice:** The worst-case time complexity of Prepend with Count = 1 and Delete_First with Count=1 for Containers.Vectors should be *O*(*N* log *N*).

258.b/2 **Reason:** We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation that takes *O*(*N*) time to access elements, that program could be unusable when the vectors are large. We allow *O*(log *N*) access because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations.

{*AI95-00302-03*} The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2). 259/2

Implementation Advice: The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2). 259.a/2

Ramification: In other words, we're requiring the use of a better than *O*(*N***2) sorting algorithm, such as Quicksort. No bubble sorts allowed! 259.b/2

{*AI95-00302-03*} Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements. 260/2

Implementation Advice: Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements. 260.a/2

To be honest: We do not mean “absolutely minimize” here; we're not intending to require a single copy for each element. Rather, we want to suggest that the sorting algorithm chosen is one that does not copy items unnecessarily. Bubble sort would not meet this advice, for instance. 260.b/2

{*AI95-00302-03*} Move should not copy elements, and should minimize copying of internal data structures. 261/2

Implementation Advice: Containers.Vectors.Move should not copy elements, and should minimize copying of internal data structures. 261.a/2

Implementation Note: Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source vector to the Target vector. 261.b/2

{*AI95-00302-03*} If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation. 262/2

Implementation Advice: If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation. 262.a/2

Reason: This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished. 262.b/2

NOTES

42 All elements of a vector occupy locations in the internal array. If a sparse container is required, a Hashed_Map should be used rather than a vector. 263/2

43 If Index_Type'Base'First = Index_Type'First an instance of Ada.Containers.Vectors will raise Constraint_Error. A value below Index_Type'First is required so that an empty vector has a meaningful value of Last_Index. 264/2

Discussion: This property is the main reason why only integer types (as opposed to any discrete type) are allowed as the index type of a vector. An enumeration or modular type would require a subtype in order to meet this requirement. 264.a/2

Extensions to Ada 95

{*AI95-00302-03*} {*extensions to Ada 95*} The package Containers.Vectors is new. 264.b/2

A.18.3 The Package Containers.Doubly_Linked_Lists

{*AI95-00302-03*} The language-defined generic package Containers.Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. A list container is optimized for insertion and deletion at any position. {*list container*} {*container (list)*} 1/2

{*AI95-00302-03*} {*node (of a list)*} A doubly-linked list container object manages a linked list of internal *nodes*, each of which contains an element and pointers to the next (successor) and previous (predecessor) internal nodes. A cursor designates a particular node within a list (and by extension the element contained in that node). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved in the container. 2/2

{*AI95-00302-03*} The *length* of a list is the number of elements it contains.{*length (of a list container)*} 3/2

Static Semantics

4/2 {*AI95-00302-03*} The generic library package Containers.Doubly_Linked_Lists has the following declaration:

5/2
```
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
   pragma Preelaborate(Doubly_Linked_Lists);
```

6/2
```
   type List is tagged private;
   pragma Preelaborable_Initialization(List);
```

7/2
```
   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);
```

8/2
```
   Empty_List : constant List;
```

9/2
```
   No_Element : constant Cursor;
```

10/2
```
   function "=" (Left, Right : List) return Boolean;
```

11/2
```
   function Length (Container : List) return Count_Type;
```

12/2
```
   function Is_Empty (Container : List) return Boolean;
```

13/2
```
   procedure Clear (Container : in out List);
```

14/2
```
   function Element (Position : Cursor)
      return Element_Type;
```

15/2
```
   procedure Replace_Element (Container : in out List;
                              Position  : in     Cursor;
                              New_Item  : in     Element_Type);
```

16/2
```
   procedure Query_Element
     (Position : in Cursor;
      Process  : not null access procedure (Element : in Element_Type));
```

17/2
```
   procedure Update_Element
     (Container : in out List;
      Position  : in     Cursor;
      Process   : not null access procedure
                    (Element : in out Element_Type));
```

18/2
```
   procedure Move (Target : in out List;
                   Source : in out List);
```

19/2
```
   procedure Insert (Container : in out List;
                     Before    : in     Cursor;
                     New_Item  : in     Element_Type;
                     Count     : in     Count_Type := 1);
```

20/2
```
   procedure Insert (Container : in out List;
                     Before    : in     Cursor;
                     New_Item  : in     Element_Type;
                     Position  :    out Cursor;
                     Count     : in     Count_Type := 1);
```

21/2
```
   procedure Insert (Container : in out List;
                     Before    : in     Cursor;
                     Position  :    out Cursor;
                     Count     : in     Count_Type := 1);
```

22/2
```
   procedure Prepend (Container : in out List;
                      New_Item  : in     Element_Type;
                      Count     : in     Count_Type := 1);
```

23/2
```
   procedure Append (Container : in out List;
                     New_Item  : in     Element_Type;
                     Count     : in     Count_Type := 1);
```

24/2
```
   procedure Delete (Container : in out List;
                     Position  : in out Cursor;
                     Count     : in     Count_Type := 1);
```

```
procedure Delete_First (Container : in out List;                      25/2
                        Count     : in     Count_Type := 1);

procedure Delete_Last (Container : in out List;                       26/2
                       Count     : in     Count_Type := 1);

procedure Reverse_Elements (Container : in out List);                 27/2

procedure Swap (Container : in out List;                              28/2
                I, J      : in     Cursor);

procedure Swap_Links (Container : in out List;                        29/2
                      I, J      : in     Cursor);

procedure Splice (Target   : in out List;                             30/2
                  Before   : in     Cursor;
                  Source   : in out List);

procedure Splice (Target   : in out List;                             31/2
                  Before   : in     Cursor;
                  Source   : in out List;
                  Position : in out Cursor);

procedure Splice (Container: in out List;                             32/2
                  Before   : in     Cursor;
                  Position : in     Cursor);

function First (Container : List) return Cursor;                      33/2

function First_Element (Container : List)                             34/2
   return Element_Type;

function Last (Container : List) return Cursor;                       35/2

function Last_Element (Container : List)                              36/2
   return Element_Type;

function Next (Position : Cursor) return Cursor;                      37/2

function Previous (Position : Cursor) return Cursor;                  38/2

procedure Next (Position : in out Cursor);                            39/2

procedure Previous (Position : in out Cursor);                        40/2

function Find (Container : List;                                      41/2
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor;

function Reverse_Find (Container : List;                              42/2
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor;

function Contains (Container : List;                                  43/2
                   Item      : Element_Type) return Boolean;

function Has_Element (Position : Cursor) return Boolean;              44/2

procedure Iterate                                                     45/2
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor));

procedure Reverse_Iterate                                             46/2
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor));

generic                                                               47/2
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
package Generic_Sorting is

   function Is_Sorted (Container : List) return Boolean;              48/2

   procedure Sort (Container : in out List);                          49/2

   procedure Merge (Target  : in out List;                            50/2
                    Source  : in out List);
```

```
51/2          end Generic_Sorting;
52/2       private
53/2          ... -- not specified by the language
54/2       end Ada.Containers.Doubly_Linked_Lists;
```

55/2 {*AI95-00302-03*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions Find, Reverse_Find, and "=" on list values return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions Find, Reverse_Find, and "=" on list values are unspecified.{*unspecified* [partial]}

55.a/2 **Ramification:** If the actual function for "=" is not symmetric and consistent, the result returned by the listed functions cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other “bad” behavior. And it can call "=" in whatever manner makes sense. But note that only the results of Find, Reverse_Find, and List "=" are unspecified; other subprograms are not allowed to break if "=" is bad (they aren't expected to use "=").

56/2 {*AI95-00302-03*} The type List is used to represent lists. The type List needs finalization (see 7.6).

57/2 {*AI95-00302-03*} Empty_List represents the empty List object. It has a length of 0. If an object of type List is not otherwise initialized, it is initialized to the same value as Empty_List.

58/2 {*AI95-00302-03*} No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

59/2 {*AI95-00302-03*} The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

60/2 {*AI95-00302-03*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

60.a/2 **Reason:** A cursor will probably be implemented in terms of one or more access values, and the effects of streaming access values is unspecified. Rather than letting the user stream junk by accident, we mandate that streaming of cursors raise Program_Error by default. The attributes can always be specified if there is a need to support streaming.

61/2 {*AI95-00302-03*} [Some operations of this generic package have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced.]

62/2 {*AI95-00302-03*} {*tamper with cursors (of a list)*} A subprogram is said to *tamper with cursors* of a list object *L* if:

63/2 • it inserts or deletes elements of *L*, that is, it calls the Insert, Clear, Delete, or Delete_Last procedures with *L* as a parameter; or

63.a/2 **To be honest:** Operations which are defined to be equivalent to a call on one of these operations also are included. Similarly, operations which call one of these as part of their definition are included.

64/2 • it reorders the elements of *L*, that is, it calls the Splice, Swap_Links, or Reverse_Elements procedures or the Sort or Merge procedures of an instance of Generic_Sorting with *L* as a parameter; or

65/2 • it finalizes *L*; or

66/2 • it calls the Move procedure with *L* as a parameter.

66.a/2 **Reason:** Swap copies elements rather than reordering them, so it doesn't tamper with cursors.

{*AI95-00302-03*} {*tamper with elements (of a list)*} A subprogram is said to *tamper with elements* of a list object *L* if: 67/2

- it tampers with cursors of *L*; or 68/2
- it replaces one or more elements of *L*, that is, it calls the Replace_Element or Swap procedures with *L* as a parameter. 69/2

Reason: Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem, so Update_Element does not cause a problem. 69.a/2

70/2
```
function "=" (Left, Right : List) return Boolean;
```

{*AI95-00302-03*} If Left and Right denote the same list object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise it returns True. Any exception raised during evaluation of element equality is propagated. 71/2

Implementation Note: This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality additional times once the answer has been determined. 71.a/2

72/2
```
function Length (Container : List) return Count_Type;
```

{*AI95-00302-03*} Returns the number of elements in Container. 73/2

74/2
```
function Is_Empty (Container : List) return Boolean;
```

{*AI95-00302-03*} Equivalent to Length (Container) = 0. 75/2

76/2
```
procedure Clear (Container : in out List);
```

{*AI95-00302-03*} Removes all the elements from Container. 77/2

78/2
```
function Element (Position : Cursor) return Element_Type;
```

{*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position. 79/2

80/2
```
procedure Replace_Element (Container : in out List;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
```

{*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns the value New_Item to the element designated by Position. 81/2

82/2
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```

{*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated. 83/2

84/2
```
procedure Update_Element
  (Container : in out List;
   Position  : in     Cursor;
   Process   : not null access procedure (Element : in out Element_Type));
```

85/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

86/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

86.a/2 **Ramification:** This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place.

87/2
```
procedure Move (Target : in out List;
                Source : in out List);
```

88/2 {*AI95-00302-03*} If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then, the nodes in Source are moved to Target (in the original order). The length of Target is set to the length of Source, and the length of Source is set to 0.

89/2
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

90/2 {*AI95-00302-03*} If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, Insert inserts Count copies of New_Item prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last node (if any). Any exception raised during allocation of internal storage is propagated, and Container is not modified.

90.a/2 **Ramification:** The check on Before checks that the cursor does not belong to some other Container. This check implies that a reference to the container is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception).

91/2
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

92/2 {*AI95-00302-03*} If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, Insert allocates Count copies of New_Item, and inserts them prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last element (if any). Position designates the first newly-inserted element. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

93/2
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

94/2 {*AI95-00302-03*} If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, Insert inserts Count new elements prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last node (if any). The new elements are initialized by default (see 3.3.1). Any exception raised during allocation of internal storage is propagated, and Container is not modified.

95/2
```
procedure Prepend (Container : in out List;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1);
```

96/2 {*AI95-00302-03*} Equivalent to Insert (Container, First (Container), New_Item, Count).

97/2
```
procedure Append (Container : in out List;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

98/2 {*AI95-00302-03*} Equivalent to Insert (Container, No_Element, New_Item, Count).

99/2
```
procedure Delete (Container : in out List;
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1);
```

100/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise Delete removes (from Container) Count elements starting at the element designated by Position (or all of the elements starting at Position if there are fewer than Count elements starting at Position). Finally, Position is set to No_Element.

101/2
```
procedure Delete_First (Container : in out List;
                        Count     : in     Count_Type := 1);
```

102/2 {*AI95-00302-03*} Equivalent to Delete (Container, First (Container), Count).

103/2
```
procedure Delete_Last (Container : in out List;
                       Count     : in     Count_Type := 1);
```

104/2 {*AI95-00302-03*} If Length (Container) <= Count then Delete_Last is equivalent to Clear (Container). Otherwise it removes the last Count nodes from Container.

105/2
```
procedure Reverse_Elements (Container : in out List);
```

106/2 {*AI95-00302-03*} Reorders the elements of Container in reverse order.

106.a/2 **Discussion:** Unlike the similar routine for a vector, elements should not be copied; rather, the nodes should be exchanged. Cursors are expected to reference the same elements afterwards.

107/2
```
procedure Swap (Container : in out List;
                I, J      : in     Cursor);
```

108/2 {*AI95-00302-03*} If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap exchanges the values of the elements designated by I and J.

108.a/2 **Ramification:** After a call to Swap, I designates the element value previously designated by J, and J designates the element value previously designated by I. The cursors do not become ambiguous from this operation.

108.b/2 **To be honest:** The implementation is not required to actually copy the elements if it can do the swap some other way. But it is allowed to copy the elements if needed.

109/2
```
procedure Swap_Links (Container : in out List;
                      I, J      : in     Cursor);
```

110/2 {*AI95-00302-03*} If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap_Links exchanges the nodes designated by I and J.

110.a/2 **Ramification:** Unlike Swap, this exchanges the nodes, not the elements. No copying is performed. I and J designate the same elements after this call as they did before it. This operation can provide better performance than Swap if the element size is large.

111/2
```
procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List);
```

112/2 {*AI95-00302-03*} If Before is not No_Element, and does not designate an element in Target, then Program_Error is propagated. Otherwise, if Source denotes the same object as Target, the operation has no effect. Otherwise, Splice reorders elements such that they are removed from Source and moved to Target, immediately prior to Before. If Before equals No_Element, the nodes of Source are spliced after the last node of Target. The length of Target is incremented by the number of nodes in Source, and the length of Source is set to 0.

113/2
```
procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List;
                  Position : in out Cursor);
```

114/2 {*AI95-00302-03*} If Position is No_Element then Constraint_Error is propagated. If Before does not equal No_Element, and does not designate an element in Target, then Program_Error is propagated. If Position does not equal No_Element, and does not designate a node in Source, then Program_Error is propagated. If Source denotes the same object as Target, then there is no effect if Position equals Before, else the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. In both cases, Position and the length of Target are unchanged. Otherwise the element designated by Position is removed from Source and moved to Target, immediately prior to Before, or, if Before equals No_Element, after the last element of Target. The length of Target is incremented, the length of Source is decremented, and Position is updated to represent an element in Target.

114.a/2 **Ramification:** If Source is the same as Target, and Position = Before, or Next(Position} = Before, Splice has no effect, as the element does not have to move to meet the postcondition.

115/2
```
procedure Splice (Container: in out List;
                  Before   : in     Cursor;
                  Position : in     Cursor);
```

116/2 {*AI95-00302-03*} If Position is No_Element then Constraint_Error is propagated. If Before does not equal No_Element, and does not designate an element in Container, then Program_Error is propagated. If Position does not equal No_Element, and does not designate a node in Container, then Program_Error is propagated. If Position equals Before there is no effect. Otherwise, the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. The length of Container is unchanged.

117/2
```
function First (Container : List) return Cursor;
```

118/2 {*AI95-00302-03*} If Container is empty, First returns the value No_Element. Otherwise it returns a cursor that designates the first node in Container.

119/2
```
function First_Element (Container : List) return Element_Type;
```

120/2 {*AI95-00302-03*} Equivalent to Element (First (Container)).

121/2
```
function Last (Container : List) return Cursor;
```

122/2 {*AI95-00302-03*} If Container is empty, Last returns the value No_Element. Otherwise it returns a cursor that designates the last node in Container.

123/2
```
function Last_Element (Container : List) return Element_Type;
```

124/2 {*AI95-00302-03*} Equivalent to Element (Last (Container)).

```
function Next (Position : Cursor) return Cursor;
```
125/2

{*AI95-00302-03*} If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the successor of the element designated by Position. 126/2

```
function Previous (Position : Cursor) return Cursor;
```
127/2

{*AI95-00302-03*} If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the predecessor of the element designated by Position. 128/2

```
procedure Next (Position : in out Cursor);
```
129/2

{*AI95-00302-03*} Equivalent to Position := Next (Position). 130/2

```
procedure Previous (Position : in out Cursor);
```
131/2

{*AI95-00302-03*} Equivalent to Position := Previous (Position). 132/2

```
function Find (Container : List;
               Item      : Element_Type;
               Position  : Cursor := No_Element)
  return Cursor;
```
133/2

{*AI95-00302-03*} If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, or at the first element if Position equals No_Element. It proceeds towards Last (Container). If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 134/2

```
function Reverse_Find (Container : List;
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor;
```
135/2

{*AI95-00302-03*} If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, or at the last element if Position equals No_Element. It proceeds towards First (Container). If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 136/2

```
function Contains (Container : List;
                   Item      : Element_Type) return Boolean;
```
137/2

{*AI95-00302-03*} Equivalent to Find (Container, Item) /= No_Element. 138/2

```
function Has_Element (Position : Cursor) return Boolean;
```
139/2

{*AI95-00302-03*} Returns True if Position designates an element, and returns False otherwise. 140/2

To be honest: This function may not detect cursors that designate deleted elements; such cursors are invalid (see below) and the result of Has_Element for an invalid cursor is unspecified (but not erroneous). 140.a/2

```
procedure Iterate
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor));
```
141/2

{*AI95-00302-03*} Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor as per the Next function. Program_Error is 142/2 propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

142.a/2 **Implementation Note:** The purpose of the tamper with cursors check is to prevent erroneous execution from the Position parameter of Process.**all** becoming invalid. This check takes place when the operations that tamper with the cursors of the container are called. The check cannot be made later (say in the body of Iterate), because that could cause the Position cursor to be invalid and potentially cause execution to become erroneous -- defeating the purpose of the check.

142.b/2 See Iterate for vectors (A.18.2) for a suggested implementation of the check.

143/2
```
procedure Reverse_Iterate
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor));
```

144/2 {*AI95-00302-03*} Iterates over the nodes in Container as per Iterate, except that elements are traversed in reverse order, starting with the last node and moving the cursor as per the Previous function.

145/2 The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.{*unspecified* [partial]}

146/2
```
function Is_Sorted (Container : List) return Boolean;
```

147/2 {*AI95-00302-03*} Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

148/2
```
procedure Sort (Container : in out List);
```

149/2 {*AI95-00302-03*} Reorders the nodes of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. The sort is stable. Any exception raised during evaluation of "<" is propagated.

149.a/2 **Ramification:** Unlike array sorts, we do require stable sorts here. That's because algorithms in the merge sort family (as described by Knuth) can be both fast and stable. Such sorts use the extra memory as offered by the links to provide better performance.

149.b/2 Note that list sorts never copy elements; it is the nodes, not the elements, that are reordered.

150/2
```
procedure Merge (Target  : in out List;
                 Source  : in out List);
```

151/2 {*AI95-00302-03*} Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

151.a/2 **Ramification:** It is a bounded error if either of the lists is unsorted, see below. The bounded error can be recovered by sorting Target after the merge call, or the lists can be pretested with Is_Sorted.

Bounded (Run-Time) Errors

152/2 {*AI95-00302-03*} {*bounded error (cause)* [partial]} Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

Erroneous Execution

153/2 {*AI95-00302-03*} A Cursor value is *invalid* if any of the following have occurred since it was created:{*invalid cursor (of a list container)*} {*cursor (invalid)* [partial]}

- 154/2 The list that contains the element it designates has been finalized;
- 155/2 The list that contains the element it designates has been used as the Source or Target of a call to Move; or
- 156/2 The element it designates has been deleted.

157/2 {*AI95-00302-03*} The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Doubly_Linked_Lists is called with an invalid cursor parameter. {*unspecified* [partial]} {*erroneous execution (cause)* [partial]}

157.a/2 **Discussion:** The list above is intended to be exhaustive. In other cases, a cursor value continues to designate its original element. For instance, cursor values survive the insertion and deletion of other nodes.

157.b/2 While it is possible to check for these cases, in many cases the overhead necessary to make the check is substantial in time or space. Implementations are encouraged to check for as many of these cases as possible and raise Program_Error if detected.

Implementation Requirements

158/2 {*AI95-00302-03*} No storage associated with a doubly-linked List object shall be lost upon assignment or scope exit.

159/2 {*AI95-00302-03*} The execution of an assignment_statement for a list shall have the effect of copying the elements from the source list object to the target list object.

159.a/2 **Implementation Note:** An assignment of a List is a “deep” copy; that is the elements are copied as well as the data structures. We say “effect of” in order to allow the implementation to avoid copying elements immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed.

Implementation Advice

160/2 {*AI95-00302-03*} Containers.Doubly_Linked_Lists should be implemented similarly to a linked list. In particular, if *N* is the length of a list, then the worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 should be *O*(log *N*).

160.a/2 **Implementation Advice:** The worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 for Containers.Doubly_Linked_Lists should be *O*(log *N*).

160.b/2 **Reason:** We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation that takes *O*(*N*) time to access elements, that program could be unusable when the lists are large. We allow *O*(log *N*) access because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations.

161/2 {*AI95-00302-03*} The worst-case time complexity of a call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2).

161.a/2 **Implementation Advice:** a call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2).

161.b/2 **Ramification:** In other words, we're requiring the use of a better than *O*(*N***2) sorting algorithm, such as Quicksort. No bubble sorts allowed!

162/2 {*AI95-00302-03*} Move should not copy elements, and should minimize copying of internal data structures.

162.a/2 **Implementation Advice:** Containers.Doubly_Link_Lists.Move should not copy elements, and should minimize copying of internal data structures.

162.b/2 **Implementation Note:** Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source container to the Target container.

163/2 {*AI95-00302-03*} If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation.

163.a/2 **Implementation Advice:** If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation.

163.b/2 **Reason:** This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished.

NOTES

164/2 44 {*AI95-00302-03*} Sorting a list never copies elements, and is a stable sort (equal elements remain in the original order). This is different than sorting an array or vector, which may need to copy elements, and is probably not a stable sort.

Extensions to Ada 95

164.a/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Doubly_Linked_Lists is new.

A.18.4 Maps

1/2 {*AI95-00302-03*} The language-defined generic packages Containers.Hashed_Maps and Containers.Ordered_Maps provide private types Map and Cursor, and a set of operations for each type. A map container allows an arbitrary type to be used as a key to find the element associated with that key. A hashed map uses a hash function to organize the keys, while an ordered map orders the keys per a specified relation. {*map container*} {*container (map)*}

2/2 {*AI95-00302-03*} This section describes the declarations that are common to both kinds of maps. See A.18.5 for a description of the semantics specific to Containers.Hashed_Maps and A.18.6 for a description of the semantics specific to Containers.Ordered_Maps.

Static Semantics

3/2 {*AI95-00302-03*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on map values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on map values are unspecified.{*unspecified* [partial]}

3.a/2 **Ramification:** If the actual function for "=" is not symmetric and consistent, the result returned by "=" for Map objects cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other “bad” behavior. And it can call "=" in whatever manner makes sense. But note that only the result of "=" for Map objects is unspecified; other subprograms are not allowed to break if "=" is bad (they aren't expected to use "=").

4/2 {*AI95-00302-03*} The type Map is used to represent maps. The type Map needs finalization (see 7.6).

5/2 {*AI95-00302-03*} {*node (of a map)*} A map contains pairs of keys and elements, called *nodes*. Map cursors designate nodes, but also can be thought of as designating an element (the element contained in the node) for consistency with the other containers. There exists an equivalence relation on keys, whose definition is different for hashed maps and ordered maps. A map never contains two or more nodes with equivalent keys. The *length* of a map is the number of nodes it contains.{*length (of a map)*}

6/2 {*AI95-00302-03*} {*first node (of a map)*} {*last node (of a map)*} {*successor node (of a map)*} Each nonempty map has two particular nodes called the *first node* and the *last node* (which may be the same). Each node except for the last node has a *successor node*. If there are no other intervening operations, starting with the first node and repeatedly going to the successor node will visit each node in the map exactly once until the last node is reached. The exact definition of these terms is different for hashed maps and ordered maps.

{*AI95-00302-03*} [Some operations of these generic packages have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced.] 7/2

{*AI95-00302-03*} {*tamper with cursors (of a map)*} A subprogram is said to *tamper with cursors* of a map object *M* if: 8/2

- it inserts or deletes elements of *M*, that is, it calls the Insert, Include, Clear, Delete, or Exclude procedures with *M* as a parameter; or 9/2

  To be honest: Operations which are defined to be equivalent to a call on one of these operations also are included. Similarly, operations which call one of these as part of their definition are included. 9.a/2

- it finalizes *M*; or 10/2
- it calls the Move procedure with *M* as a parameter; or 11/2
- it calls one of the operations defined to tamper with the cursors of *M*. 12/2

  Ramification: Replace only modifies a key and element rather than rehashing, so it does not tamper with cursors. 12.a/2

{*AI95-00302-03*} {*tamper with elements (of a map)*} A subprogram is said to *tamper with elements* of a map object *M* if: 13/2

- it tampers with cursors of *M*; or 14/2
- it replaces one or more elements of *M*, that is, it calls the Replace or Replace_Element procedures with *M* as a parameter. 15/2

  Reason: Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem, so Update_Element does not cause a problem. 15.a/2

{*AI95-00302-03*} Empty_Map represents the empty Map object. It has a length of 0. If an object of type Map is not otherwise initialized, it is initialized to the same value as Empty_Map. 16/2

{*AI95-00302-03*} No_Element represents a cursor that designates no node. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element. 17/2

{*AI95-00302-03*} The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container. 18/2

{*AI95-00302-03*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error. 19/2

Reason: A cursor will probably be implemented in terms of one or more access values, and the effects of streaming access values is unspecified. Rather than letting the user stream junk by accident, we mandate that streaming of cursors raise Program_Error by default. The attributes can always be specified if there is a need to support streaming. 19.a/2

```
function "=" (Left, Right : Map) return Boolean;
```
20/2

{*AI95-00302-03*} If Left and Right denote the same map object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each key *K* in Left, the function returns False if: 21/2

- a key equivalent to *K* is not present in Right; or 22/2
- the element associated with *K* in Left is not equal to the element associated with *K* in Right (using the generic formal equality operator for elements). 23/2

If the function has not returned a result after checking all of the keys, it returns True. Any exception raised during evaluation of key equivalence or element equality is propagated. 24/2

24.a/2 **Implementation Note:** This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality additional times once the answer has been determined.

25/2
```
function Length (Container : Map) return Count_Type;
```

26/2 {*AI95-00302-03*} Returns the number of nodes in Container.

27/2
```
function Is_Empty (Container : Map) return Boolean;
```

28/2 {*AI95-00302-03*} Equivalent to Length (Container) = 0.

29/2
```
procedure Clear (Container : in out Map);
```

30/2 {*AI95-00302-03*} Removes all the nodes from Container.

31/2
```
function Key (Position : Cursor) return Key_Type;
```

32/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Key returns the key component of the node designated by Position.

33/2
```
function Element (Position : Cursor) return Element_Type;
```

34/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element component of the node designated by Position.

35/2
```
procedure Replace_Element (Container : in out Map;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
```

36/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns New_Item to the element of the node designated by Position.

37/2
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Key     : in Key_Type;
                                         Element : in Element_Type));
```

38/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

39/2
```
procedure Update_Element
  (Container : in out Map;
   Position  : in     Cursor;
   Process   : not null access procedure (Key     : in     Key_Type;
                                          Element : in out Element_Type));
```

40/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

41/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

41.a/2 **Ramification:** This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place.

42/2
```
procedure Move (Target : in out Map;
                Source : in out Map);
```

43/2 {*AI95-00302-03*} If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then, each node from Source is removed from Source and inserted into Target. The length of Source is 0 after a successful call to Move.

44/2
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

45/2 {*AI95-00302-03*} Insert checks if a node with a key equivalent to Key is already present in Container. If a match is found, Inserted is set to False and Position designates the element with the matching key. Otherwise, Insert allocates a new node, initializes it to Key and New_Item, and adds it to Container; Inserted is set to True and Position designates the newly-inserted node. Any exception raised during allocation is propagated and Container is not modified.

46/2
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

47/2 {*AI95-00302-03*} Insert inserts Key into Container as per the five-parameter Insert, with the difference that an element initialized by default (see 3.3.1) is inserted.

48/2
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type);
```

49/2 {*AI95-00302-03*} Insert inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then Constraint_Error is propagated.

49.a/2 **Ramification:** This is equivalent to:

49.b/2
```
declare
  Inserted : Boolean; C : Cursor;
begin
  Insert (Container, Key, New_Item, C, Inserted);
  if not Inserted then
     raise Constraint_Error;
  end if;
end;
```

49.c/2 but doesn't require the hassle of **out** parameters.

50/2
```
procedure Include (Container : in out Map;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);
```

51/2 {*AI95-00302-03*} Include inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then this operation assigns Key and New_Item to the matching node. Any exception raised during assignment is propagated.

51.a/2 **Ramification:** This is equivalent to:

51.b/2
```
declare
  C : Cursor := Find (Container, Key);
begin
  if C = No_Element then
     Insert (Container, Key, New_Item);
  else
     Replace (Container, Key, New_Item);
  end if;
end;
```

51.c/2 but this avoids doing the search twice.

52/2
```
procedure Replace (Container : in out Map;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);
```

53/2 {*AI95-00302-03*} Replace checks if a node with a key equivalent to Key is present in Container. If a match is found, Replace assigns Key and New_Item to the matching node; otherwise, Constraint_Error is propagated.

53.a/2 **Discussion:** We update the key as well as the element, as the key might include additional information that does not participate in equivalence. If only the element needs to be updated, use Replace_Element (Find (Container, Key), New_Element).

54/2
```
procedure Exclude (Container : in out Map;
                   Key       : in     Key_Type);
```

55/2 {*AI95-00302-03*} Exclude checks if a node with a key equivalent to Key is present in Container. If a match is found, Exclude removes the node from the map.

55.a/2 **Ramification:** Exclude should work on an empty map; nothing happens in that case.

56/2
```
procedure Delete (Container : in out Map;
                  Key       : in     Key_Type);
```

57/2 {*AI95-00302-03*} Delete checks if a node with a key equivalent to Key is present in Container. If a match is found, Delete removes the node from the map; otherwise, Constraint_Error is propagated.

58/2
```
procedure Delete (Container : in out Map;
                  Position  : in out Cursor);
```

59/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete removes the node designated by Position from the map. Position is set to No_Element on return.

59.a/2 **Ramification:** The check on Position checks that the cursor does not belong to some other map. This check implies that a reference to the map is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception).

60/2
```
function First (Container : Map) return Cursor;
```

61/2 {*AI95-00302-03*} If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first node in Container.

62/2
```
function Next (Position  : Cursor) return Cursor;
```

63/2 {*AI95-00302-03*} Returns a cursor that designates the successor of the node designated by Position. If Position designates the last node, then No_Element is returned. If Position equals No_Element, then No_Element is returned.

64/2
```
procedure Next (Position  : in out Cursor);
```

65/2 {*AI95-00302-03*} Equivalent to Position := Next (Position).

66/2
```
function Find (Container : Map;
               Key       : Key_Type) return Cursor;
```

67/2 {*AI95-00302-03*} If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if a node with a key equivalent to Key is present in Container. If a match is found, a cursor designating the matching node is returned; otherwise, No_Element is returned.

```
function Element (Container : Map;
                  Key       : Key_Type) return Element_Type;
```
68/2

{*AI95-00302-03*} Equivalent to Element (Find (Container, Key)). 69/2

```
function Contains (Container : Map;
                   Key       : Key_Type) return Boolean;
```
70/2

{*AI95-00302-03*} Equivalent to Find (Container, Key) /= No_Element. 71/2

```
function Has_Element (Position : Cursor) return Boolean;
```
72/2

{*AI95-00302-03*} Returns True if Position designates a node, and returns False otherwise. 73/2

To be honest: This function may not detect cursors that designate deleted elements; such cursors are invalid (see below); the result of Has_Element for invalid cursors is unspecified (but not erroneous). 73.a/2

```
procedure Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor));
```
74/2

{*AI95-00302-03*} Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor according to the successor relation. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated. 75/2

Implementation Note: The "tamper with cursors" check takes place when the operations that insert or delete elements, and so on, are called. 75.a/2

See Iterate for vectors (A.18.2) for a suggested implementation of the check. 75.b/2

Erroneous Execution

{*AI95-00302-03*} A Cursor value is *invalid* if any of the following have occurred since it was created:{*invalid cursor (of a map)*} {*cursor (invalid)* [partial]} 76/2

- The map that contains the node it designates has been finalized; 77/2
- The map that contains the node it designates has been used as the Source or Target of a call to Move; or 78/2
- The node it designates has been deleted from the map. 79/2

The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter.{*unspecified* [partial]} Execution is erroneous if any other subprogram declared in Containers.Hashed_Maps or Containers.Ordered_Maps is called with an invalid cursor parameter.{*erroneous execution (cause)* [partial]} 80/2

Discussion: The list above is intended to be exhaustive. In other cases, a cursor value continues to designate its original element. For instance, cursor values survive the insertion and deletion of other nodes. 80.a/2

While it is possible to check for these cases, in many cases the overhead necessary to make the check is substantial in time or space. Implementations are encouraged to check for as many of these cases as possible and raise Program_Error if detected. 80.b/2

Implementation Requirements

{*AI95-00302-03*} No storage associated with a Map object shall be lost upon assignment or scope exit. 81/2

{*AI95-00302-03*} The execution of an assignment_statement for a map shall have the effect of copying the elements from the source map object to the target map object. 82/2

Implementation Note: An assignment of a Map is a "deep" copy; that is the elements are copied as well as the data structures. We say "effect of" in order to allow the implementation to avoid copying elements immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed. 82.a/2

Implementation Advice

83/2 {*AI95-00302-03*} Move should not copy elements, and should minimize copying of internal data structures.

83.a/2 **Implementation Advice:** Move for a map should not copy elements, and should minimize copying of internal data structures.

83.b/2 **Implementation Note:** Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source container to the Target container.

84/2 {*AI95-00302-03*} If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation.

84.a/2 **Implementation Advice:** If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation.

84.b/2 **Reason:** This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished.

Wording Changes from Ada 95

84.c/2 {*AI95-00302-03*} This description of maps is new; the extensions are documented with the specific packages.

A.18.5 The Package Containers.Hashed_Maps

Static Semantics

1/2 {*AI95-00302-03*} The generic library package Containers.Hashed_Maps has the following declaration:

2/2
```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function Hash (Key : Key_Type) return Hash_Type;
   with function Equivalent_Keys (Left, Right : Key_Type)
      return Boolean;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Hashed_Maps is
   pragma Preelaborate(Hashed_Maps);
```

3/2
```
   type Map is tagged private;
   pragma Preelaborable_Initialization(Map);
```

4/2
```
   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);
```

5/2
```
   Empty_Map : constant Map;
```

6/2
```
   No_Element : constant Cursor;
```

7/2
```
   function "=" (Left, Right : Map) return Boolean;
```

8/2
```
   function Capacity (Container : Map) return Count_Type;
```

9/2
```
   procedure Reserve_Capacity (Container : in out Map;
                               Capacity  : in     Count_Type);
```

10/2
```
   function Length (Container : Map) return Count_Type;
```

11/2
```
   function Is_Empty (Container : Map) return Boolean;
```

12/2
```
   procedure Clear (Container : in out Map);
```

13/2
```
   function Key (Position : Cursor) return Key_Type;
```

14/2
```
   function Element (Position : Cursor) return Element_Type;
```

15/2
```
   procedure Replace_Element (Container : in out Map;
                              Position  : in     Cursor;
                              New_Item  : in     Element_Type);
```

16/2
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Key     : in Key_Type;
                                         Element : in Element_Type));
```

17/2
```
procedure Update_Element
  (Container : in out Map;
   Position  : in     Cursor;
   Process   : not null access procedure
                   (Key     : in     Key_Type;
                    Element : in out Element_Type));
```

18/2
```
procedure Move (Target : in out Map;
                Source : in out Map);
```

19/2
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

20/2
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

21/2
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type);
```

22/2
```
procedure Include (Container : in out Map;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);
```

23/2
```
procedure Replace (Container : in out Map;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);
```

24/2
```
procedure Exclude (Container : in out Map;
                   Key       : in     Key_Type);
```

25/2
```
procedure Delete (Container : in out Map;
                  Key       : in     Key_Type);
```

26/2
```
procedure Delete (Container : in out Map;
                  Position  : in out Cursor);
```

27/2
```
function First (Container : Map)
   return Cursor;
```

28/2
```
function Next (Position  : Cursor) return Cursor;
```

29/2
```
procedure Next (Position  : in out Cursor);
```

30/2
```
function Find (Container : Map;
               Key       : Key_Type)
   return Cursor;
```

31/2
```
function Element (Container : Map;
                  Key       : Key_Type)
   return Element_Type;
```

32/2
```
function Contains (Container : Map;
                   Key       : Key_Type) return Boolean;
```

33/2
```
function Has_Element (Position : Cursor) return Boolean;
```

34/2
```
function Equivalent_Keys (Left, Right : Cursor)
   return Boolean;
```

35/2
```
function Equivalent_Keys (Left  : Cursor;
                          Right : Key_Type)
   return Boolean;
```

36/2
```
function Equivalent_Keys (Left  : Key_Type;
                          Right : Cursor)
   return Boolean;
```

37/2
```
      procedure Iterate
        (Container : in Map;
         Process   : not null access procedure (Position : in Cursor));
```
38/2
```
   private
```
39/2
```
      ... -- not specified by the language
```
40/2
```
   end Ada.Containers.Hashed_Maps;
```

41/2 {*AI95-00302-03*} An object of type Map contains an expandable hash table, which is used to provide direct access to nodes. The *capacity* of an object of type Map is the maximum number of nodes that can be inserted into the hash table prior to it being automatically expanded.{*capacity (of a hashed map)*}

41.a/2 **Implementation Note:** The expected implementation for a Map uses a hash table which is grown when it is too small, with linked lists hanging off of each bucket. Note that in that implementation a cursor needs a back pointer to the Map object to implement iteration; that could either be in the nodes, or in the cursor object. To provide an average *O*(1) access time, capacity would typically equal the number of buckets in such an implementation, so that the average bucket linked list length would be no more than 1.0.

41.b/2 There is no defined relationship between elements in a hashed map. Typically, iteration will return elements in the order that they are hashed in.

42/2 {*AI95-00302-03*} {*equivalent key (of a hashed map)*} Two keys *K1* and *K2* are defined to be *equivalent* if Equivalent_Keys (*K1*, *K2*) returns True.

43/2 {*AI95-00302-03*} The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular key value. For any two equivalent key values, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.{*unspecified* [partial]}

43.a/2 **Implementation Note:** The implementation is not required to protect against Hash raising an exception, or returning random numbers, or any other “bad” behavior. It's not practical to do so, and a broken Hash function makes the container unusable.

43.b/2 The implementation can call Hash whenever it is needed; we don't want to specify how often that happens. The result must remain the same (this is logically a pure function), or the behavior is unspecified.

44/2 {*AI95-00302-03*} The actual function for the generic formal function Equivalent_Keys on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Keys behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Keys, and how many times they call it, is unspecified.{*unspecified* [partial]}

44.a/2 **Implementation Note:** As with Hash, the implementation is not required to protect against Equivalent_Keys raising an exception or returning random results. Similarly, the implementation can call this operation whenever it is needed. The result must remain the same (this is a logically pure function), or the behavior is unspecified.

45/2 {*AI95-00302-03*} If the value of a key stored in a node of a map is changed other than by an operation in this package such that at least one of Hash or Equivalent_Keys give different results, the behavior of this package is unspecified.{*unspecified* [partial]}

45.a/2 **Implementation Note:** The implementation is not required to protect against changes to key values other than via the operations declared in the Hashed_Maps package.

45.b/2 To see how this could happen, imagine an instance of Hashed_Maps where the key type is an access-to-variable type and Hash returns a value derived from the components of the designated object. Then, any operation that has a key value could modify those components and change the hash value:

45.c/2
```
    Key (Map).Some_Component := New_Value;
```

45.d/2 This is really a design error on the part of the user of the map; it shouldn't be possible to modify keys stored in a map. But we can't prevent this error anymore than we can prevent someone passing as Hash a random number generator.

{*AI95-00302-03*} {*first node (of a hashed map)*} {*last node (of a hashed map)*} {*successor node (of a hashed map)*} Which nodes are the first node and the last node of a map, and which node is the successor of a given node, are unspecified, other than the general semantics described in A.18.4.{*unspecified* [partial]} 46/2

Implementation Note: Typically the first node will be the first node in the first bucket, the last node will be the last node in the last bucket, and the successor will be obtained by following the collision list, and going to the next bucket at the end of each bucket. 46.a/2

```
function Capacity (Container : Map) return Count_Type;
```
47/2

{*AI95-00302-03*} Returns the capacity of Container. 48/2

```
procedure Reserve_Capacity (Container : in out Map;
                            Capacity  : in     Count_Type);
```
49/2

{*AI95-00302-03*} Reserve_Capacity allocates a new hash table such that the length of the resulting map can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the nodes in Container onto the new hash table. It replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified. 50/2

Reserve_Capacity tampers with the cursors of Container. 51/2

Implementation Note: This routine is used to preallocate the internal hash table to the specified capacity such that future Inserts do not require expansion of the hash table. Therefore, the implementation should allocate the needed memory to make that true at this point, even though the visible semantics could be preserved by waiting until enough elements are inserted. 51.a/2

While Reserve_Capacity can be used to reduce the capacity of a map, we do not specify whether an implementation actually supports reduction of the capacity. Since the actual capacity can be anything greater than or equal to Count, an implementation never has to reduce the capacity. 51.b/2

Reserve_Capacity tampers with the cursors, as rehashing probably will change the order that elements are stored in the map. 51.c/2

```
procedure Clear (Container : in out Map);
```
52/2

{*AI95-00302-03*} In addition to the semantics described in A.18.4, Clear does not affect the capacity of Container. 53/2

Implementation Note: In: 53.a/2

```
procedure Move (Target : in out Map;
                Source : in out Map);
```
53.b/2

The intended implementation is that the internal hash table of Target is first deallocated; then the internal hash table is removed from Source and moved to Target. 53.c/2

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```
54/2

{*AI95-00302-03*} In addition to the semantics described in A.18.4, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value. 55/2

Implementation Note: Insert should only compare keys that hash to the same bucket in the hash table. 55.a/2

We specify when Reserve_Capacity is called to bound the overhead of capacity expansion operations (which are potentially expensive). Moreover, expansion can be predicted by comparing Capacity(Map) to Length(Map). Since we don't specify by how much the hash table is expanded, this only can be used to predict the next expansion, not later ones. 55.b/2

55.c/2 **Implementation Note:** In:

55.d/2
```
procedure Exclude (Container : in out Map;
                   Key       : in     Key_Type);
```

55.e/2 Exclude should only compare keys that hash to the same bucket in the hash table.

55.f/2 **Implementation Note:** In:

55.g/2
```
procedure Delete (Container : in out Map;
                  Key       : in     Key_Type);
```

55.h/2 Delete should only compare keys that hash to the same bucket in the hash table. The node containing the element may be deallocated now, or it may be saved and reused later.

55.i/2 **Implementation Note:** In:

55.j/2
```
function First (Container : Map) return Cursor;
```

55.k/2 In a typical implementation, this will be the first node in the lowest numbered hash bucket that contains a node.

55.l/2 **Implementation Note:** In:

55.m/2
```
function Next (Position  : Cursor) return Cursor;
```

55.n/2 In a typical implementation, this will return the next node in a bucket; if Position is the last node in a bucket, this will return the first node in the next non-empty bucket.

55.o/2 A typical implementation will need to a keep a pointer at the map container in the cursor in order to implement this function.

55.p/2 **Implementation Note:** In:

55.q/2
```
function Find (Container : Map;
               Key       : Key_Type) return Cursor;
```

55.r/2 Find should only compare keys that hash to the same bucket in the hash table.

56/2
```
function Equivalent_Keys (Left, Right : Cursor)
      return Boolean;
```

57/2 {*AI95-00302-03*} Equivalent to Equivalent_Keys (Key (Left), Key (Right)).

58/2
```
function Equivalent_Keys (Left  : Cursor;
                          Right : Key_Type) return Boolean;
```

59/2 {*AI95-00302-03*} Equivalent to Equivalent_Keys (Key (Left), Right).

60/2
```
function Equivalent_Keys (Left  : Key_Type;
                          Right : Cursor) return Boolean;
```

61/2 {*AI95-00302-03*} Equivalent to Equivalent_Keys (Left, Key (Right)).

Implementation Advice

62/2 {*AI95-00302-03*} If *N* is the length of a map, the average time complexity of the subprograms Element, Insert, Include, Replace, Delete, Exclude and Find that take a key parameter should be *O*(log *N*). The average time complexity of the subprograms that take a cursor parameter should be *O*(1). The average time complexity of Reserve_Capacity should be *O*(*N*).

62.a/2 **Implementation Advice:** The average time complexity of Element, Insert, Include, Replace, Delete, Exclude and Find operations that take a key parameter for Containers.Hashed_Maps should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Maps that take a cursor parameter should be *O*(1).

62.b/2 **Reason:** We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation for which Find is *O*(*N*), that program could be unusable when the maps are large. We allow *O*(log *N*) access because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations.

Extensions to Ada 95

62.c/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Hashed_Maps is new.

A.18.6 The Package Containers.Ordered_Maps

Static Semantics

{*AI95-00302-03*} The generic library package Containers.Ordered_Maps has the following declaration: 1/2

2/2
```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function "<" (Left, Right : Key_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Maps is
   pragma Preelaborate(Ordered_Maps);
```

3/2
```
   function Equivalent_Keys (Left, Right : Key_Type) return Boolean;
```

4/2
```
   type Map is tagged private;
   pragma Preelaborable_Initialization(Map);
```

5/2
```
   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);
```

6/2
```
   Empty_Map : constant Map;
```

7/2
```
   No_Element : constant Cursor;
```

8/2
```
   function "=" (Left, Right : Map) return Boolean;
```

9/2
```
   function Length (Container : Map) return Count_Type;
```

10/2
```
   function Is_Empty (Container : Map) return Boolean;
```

11/2
```
   procedure Clear (Container : in out Map);
```

12/2
```
   function Key (Position : Cursor) return Key_Type;
```

13/2
```
   function Element (Position : Cursor) return Element_Type;
```

14/2
```
   procedure Replace_Element (Container : in out Map;
                              Position  : in     Cursor;
                              New_Item  : in     Element_Type);
```

15/2
```
   procedure Query_Element
     (Position : in Cursor;
      Process  : not null access procedure (Key     : in Key_Type;
                                            Element : in Element_Type));
```

16/2
```
   procedure Update_Element
     (Container : in out Map;
      Position  : in     Cursor;
      Process   : not null access procedure
                      (Key     : in     Key_Type;
                       Element : in out Element_Type));
```

17/2
```
   procedure Move (Target : in out Map;
                   Source : in out Map);
```

18/2
```
   procedure Insert (Container : in out Map;
                     Key       : in     Key_Type;
                     New_Item  : in     Element_Type;
                     Position  :    out Cursor;
                     Inserted  :    out Boolean);
```

19/2
```
   procedure Insert (Container : in out Map;
                     Key       : in     Key_Type;
                     Position  :    out Cursor;
                     Inserted  :    out Boolean);
```

20/2
```
   procedure Insert (Container : in out Map;
                     Key       : in     Key_Type;
                     New_Item  : in     Element_Type);
```

21/2
```
   procedure Include (Container : in out Map;
                      Key       : in     Key_Type;
                      New_Item  : in     Element_Type);
```

```
22/2       procedure Replace (Container : in out Map;
                              Key       : in     Key_Type;
                              New_Item  : in     Element_Type);
23/2       procedure Exclude (Container : in out Map;
                              Key       : in     Key_Type);
24/2       procedure Delete (Container : in out Map;
                             Key       : in     Key_Type);
25/2       procedure Delete (Container : in out Map;
                             Position  : in out Cursor);
26/2       procedure Delete_First (Container : in out Map);
27/2       procedure Delete_Last (Container : in out Map);
28/2       function First (Container : Map) return Cursor;
29/2       function First_Element (Container : Map) return Element_Type;
30/2       function First_Key (Container : Map) return Key_Type;
31/2       function Last (Container : Map) return Cursor;
32/2       function Last_Element (Container : Map) return Element_Type;
33/2       function Last_Key (Container : Map) return Key_Type;
34/2       function Next (Position : Cursor) return Cursor;
35/2       procedure Next (Position : in out Cursor);
36/2       function Previous (Position : Cursor) return Cursor;
37/2       procedure Previous (Position : in out Cursor);
38/2       function Find (Container : Map;
                          Key       : Key_Type) return Cursor;
39/2       function Element (Container : Map;
                             Key       : Key_Type) return Element_Type;
40/2       function Floor (Container : Map;
                           Key       : Key_Type) return Cursor;
41/2       function Ceiling (Container : Map;
                             Key       : Key_Type) return Cursor;
42/2       function Contains (Container : Map;
                              Key       : Key_Type) return Boolean;
43/2       function Has_Element (Position : Cursor) return Boolean;
44/2       function "<" (Left, Right : Cursor) return Boolean;
45/2       function ">" (Left, Right : Cursor) return Boolean;
46/2       function "<" (Left : Cursor; Right : Key_Type) return Boolean;
47/2       function ">" (Left : Cursor; Right : Key_Type) return Boolean;
48/2       function "<" (Left : Key_Type; Right : Cursor) return Boolean;
49/2       function ">" (Left : Key_Type; Right : Cursor) return Boolean;
50/2       procedure Iterate
             (Container : in Map;
              Process   : not null access procedure (Position : in Cursor));
51/2       procedure Reverse_Iterate
             (Container : in Map;
              Process   : not null access procedure (Position : in Cursor));
52/2    private
53/2       ... -- not specified by the language
54/2    end Ada.Containers.Ordered_Maps;
```

{*AI95-00302-03*} {*equivalent key (of an ordered map)*} Two keys *K1* and *K2* are *equivalent* if both *K1* < *K2* and *K2* < *K1* return False, using the generic formal "<" operator for keys. Function Equivalent_Keys returns True if Left and Right are equivalent, and False otherwise. 55/2

{*AI95-00302-03*} The actual function for the generic formal function "<" on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive. If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.{*unspecified* [partial]} 56/2

Implementation Note: The implementation is not required to protect against "<" raising an exception, or returning random results, or any other “bad” behavior. It's not practical to do so, and a broken "<" function makes the container unusable. 56.a/2

The implementation can call "<" whenever it is needed; we don't want to specify how often that happens. The result must remain the same (this is a logically pure function), or the behavior is unspecified. 56.b/2

{*AI95-00302-03*} If the value of a key stored in a map is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.{*unspecified* [partial]} 57/2

Implementation Note: The implementation is not required to protect against changes to key values other than via the operations declared in the Ordered_Maps package. 57.a/2

To see how this could happen, imagine an instance of Ordered_Maps package where the key type is an access-to-variable type and "<" returns a value derived from comparing the components of the designated objects. Then, any operation that has a key value (even if the key value is constant) could modify those components and change the result of "<": 57.b/2

```
    Key (Map).Some_Component := New_Value;
```
57.c/2

This is really a design error on the part of the user of the map; it shouldn't be possible to modify keys stored in a map such that "<" changes. But we can't prevent this error anymore than we can prevent someone passing as "<" a routine that produces random answers. 57.d/2

{*AI95-00302-03*} {*first node (of an ordered map)*} {*last node (of an ordered map)*} {*successor node (of an ordered map)*} The first node of a nonempty map is the one whose key is less than the key of all the other nodes in the map. The last node of a nonempty map is the one whose key is greater than the key of all the other elements in the map. The successor of a node is the node with the smallest key that is larger than the key of the given node. The predecessor of a node is the node with the largest key that is smaller than the key of the given node. All comparisons are done using the generic formal "<" operator for keys. 58/2

```
procedure Delete_First (Container : in out Map);
```
59/2

{*AI95-00302-03*} If Container is empty, Delete_First has no effect. Otherwise the node designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container. 60/2

```
procedure Delete_Last (Container : in out Map);
```
61/2

{*AI95-00302-03*} If Container is empty, Delete_Last has no effect. Otherwise the node designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container. 62/2

```
function First_Element (Container : Map) return Element_Type;
```
63/2

{*AI95-00302-03*} Equivalent to Element (First (Container)). 64/2

```
function First_Key (Container : Map) return Key_Type;
```
65/2

{*AI95-00302-03*} Equivalent to Key (First (Container)). 66/2

67/2
```
function Last (Container : Map) return Cursor;
```

68/2 {*AI95-00302-03*} Returns a cursor that designates the last node in Container. If Container is empty, returns No_Element.

69/2
```
function Last_Element (Container : Map) return Element_Type;
```

70/2 {*AI95-00302-03*} Equivalent to Element (Last (Container)).

71/2
```
function Last_Key (Container : Map) return Key_Type;
```

72/2 {*AI95-00302-03*} Equivalent to Key (Last (Container)).

73/2
```
function Previous (Position : Cursor) return Cursor;
```

74/2 {*AI95-00302-03*} If Position equals No_Element, then Previous returns No_Element. Otherwise Previous returns a cursor designating the node that precedes the one designated by Position. If Position designates the first element, then Previous returns No_Element.

75/2
```
procedure Previous (Position : in out Cursor);
```

76/2 {*AI95-00302-03*} Equivalent to Position := Previous (Position).

77/2
```
function Floor (Container : Map;
                Key       : Key_Type) return Cursor;
```

78/2 {*AI95-00302-03*} Floor searches for the last node whose key is not greater than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise No_Element is returned.

79/2
```
function Ceiling (Container : Map;
                  Key       : Key_Type) return Cursor;
```

80/2 {*AI95-00302-03*} Ceiling searches for the first node whose key is not less than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise No_Element is returned.

81/2
```
function "<" (Left, Right : Cursor) return Boolean;
```

82/2 {*AI95-00302-03*} Equivalent to Key (Left) < Key (Right).

83/2
```
function ">" (Left, Right : Cursor) return Boolean;
```

84/2 {*AI95-00302-03*} Equivalent to Key (Right) < Key (Left).

85/2
```
function "<" (Left : Cursor; Right : Key_Type) return Boolean;
```

86/2 {*AI95-00302-03*} Equivalent to Key (Left) < Right.

87/2
```
function ">" (Left : Cursor; Right : Key_Type) return Boolean;
```

88/2 {*AI95-00302-03*} Equivalent to Right < Key (Left).

89/2
```
function "<" (Left : Key_Type; Right : Cursor) return Boolean;
```

90/2 {*AI95-00302-03*} Equivalent to Left < Key (Right).

91/2
```
function ">" (Left : Key_Type; Right : Cursor) return Boolean;
```

92/2 {*AI95-00302-03*} Equivalent to Key (Right) < Left.

93/2
```
procedure Reverse_Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor));
```

94/2 {*AI95-00302-03*} Iterates over the nodes in Container as per Iterate, with the difference that the nodes are traversed in predecessor order, starting with the last node.

Implementation Advice

{*AI95-00302-03*} If *N* is the length of a map, then the worst-case time complexity of the Element, Insert, Include, Replace, Delete, Exclude and Find operations that take a key parameter should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms that take a cursor parameter should be *O*(1). 95/2

Implementation Advice: The worst-case time complexity of Element, Insert, Include, Replace, Delete, Exclude and Find operations that take a key parameter for Containers.Ordered_Maps should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms of Containers.Ordered_Maps that take a cursor parameter should be *O*(1). 95.a/2

Implementation Note: A balanced (red-black) tree for keys has *O*(log *N*) worst-case performance. Note that a *O*(*N*) worst-case implementation (like a list) would be wrong. 95.b/2

Reason: We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation that takes *O*(*N*) to find elements, that program could be unusable when the maps are large. We allow the extra log *N* factors because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations. 95.c/2

Extensions to Ada 95

{*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Ordered_Maps is new. 95.d/2

A.18.7 Sets

{*AI95-00302-03*} The language-defined generic packages Containers.Hashed_Sets and Containers.Ordered_Sets provide private types Set and Cursor, and a set of operations for each type. A set container allows elements of an arbitrary type to be stored without duplication. A hashed set uses a hash function to organize elements, while an ordered set orders its element per a specified relation.{*set container*} {*container (set)*} 1/2

{*AI95-00302-03*} This section describes the declarations that are common to both kinds of sets. See A.18.8 for a description of the semantics specific to Containers.Hashed_Sets and A.18.9 for a description of the semantics specific to Containers.Ordered_Sets. 2/2

Static Semantics

{*AI95-00302-03*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on set values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on set values are unspecified.{*unspecified* [partial]} 3/2

Ramification: If the actual function for "=" is not symmetric and consistent, the result returned by the "=" for Set objects cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other “bad” behavior. And it can call "=" in whatever manner makes sense. But note that only the result of "=" for Set objects is unspecified; other subprograms are not allowed to break if "=" is bad (they aren't expected to use "="). 3.a/2

{*AI95-00302-03*} The type Set is used to represent sets. The type Set needs finalization (see 7.6). 4/2

{*AI95-00302-03*} A set contains elements. Set cursors designate elements. There exists an equivalence relation on elements, whose definition is different for hashed sets and ordered sets. A set never contains two or more equivalent elements. The *length* of a set is the number of elements it contains.{*length (of a set)*} 5/2

{*AI95-00302-03*} {*first element (of a set)*} {*last element (of a set)*} {*successor element (of a set)*} Each nonempty set has two particular elements called the *first element* and the *last element* (which may be the same). Each element except for the last element has a *successor element*. If there are no other intervening operations, starting with the first element and repeatedly going to the successor element will visit each 6/2 element in the set exactly once until the last element is reached. The exact definition of these terms is different for hashed sets and ordered sets.

7/2 {*AI95-00302-03*} [Some operations of these generic packages have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced.]

8/2 {*AI95-00302-03*} {*tamper with cursors (of a set)*} A subprogram is said to *tamper with cursors* of a set object *S* if:

9/2
- it inserts or deletes elements of *S*, that is, it calls the Insert, Include, Clear, Delete, Exclude, or Replace_Element procedures with *S* as a parameter; or

9.a/2 **To be honest:** Operations which are defined to be equivalent to a call on one of these operations also are included. Similarly, operations which call one of these as part of their definition are included.

9.b/2 **Discussion:** We have to include Replace_Element here because it might delete and reinsert the element if it moves in the set. That could change the order of iteration, which is what this check is designed to prevent. Replace is also included, as it is defined in terms of Replace_Element.

10/2
- it finalizes *S*; or

11/2
- it calls the Move procedure with *S* as a parameter; or

12/2
- it calls one of the operations defined to tamper with cursors of *S*.

13/2 {*AI95-00302-03*} {*tamper with elements (of a set)*} A subprogram is said to *tamper with elements* of a set object *S* if:

14/2
- it tampers with cursors of *S*.

14.a/2 **Reason:** Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem, so Update_Element_Preserving_Key does not cause a problem.

14.b/2 We don't need to list Replace and Replace_Element here because they are covered by “tamper with cursors”. For Set, “tamper with cursors” and “tamper with elements” are the same. We leave both terms so that the rules for routines like Iterate and Query_Element are consistent across all containers.

15/2 {*AI95-00302-03*} Empty_Set represents the empty Set object. It has a length of 0. If an object of type Set is not otherwise initialized, it is initialized to the same value as Empty_Set.

16/2 {*AI95-00302-03*} No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

17/2 {*AI95-00302-03*} The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

18/2 {*AI95-00302-03*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

18.a/2 **Reason:** A cursor will probably be implemented in terms of one or more access values, and the effects of streaming access values is unspecified. Rather than letting the user stream junk by accident, we mandate that streaming of cursors raise Program_Error by default. The attributes can always be specified if there is a need to support streaming.

19/2
```
function "=" (Left, Right : Set) return Boolean;
```

20/2 {*AI95-00302-03*} If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element *E* in Left, the function returns False if an element equal to *E* (using the generic formal equality operator) is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equality is propagated.

Implementation Note: This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality additional times once the answer has been determined. 20.a/2

```
function Equivalent_Sets (Left, Right : Set) return Boolean;
```
21/2

{*AI95-00302-03*} If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element *E* in Left, the function returns False if an element equivalent to *E* is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equivalence is propagated. 22/2

```
function To_Set (New_Item : Element_Type) return Set;
```
23/2

{*AI95-00302-03*} Returns a set containing the single element New_Item. 24/2

```
function Length (Container : Set) return Count_Type;
```
25/2

{*AI95-00302-03*} Returns the number of elements in Container. 26/2

```
function Is_Empty (Container : Set) return Boolean;
```
27/2

{*AI95-00302-03*} Equivalent to Length (Container) = 0. 28/2

```
procedure Clear (Container : in out Set);
```
29/2

{*AI95-00302-03*} Removes all the elements from Container. 30/2

```
function Element (Position : Cursor) return Element_Type;
```
31/2

{*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position. 32/2

```
procedure Replace_Element (Container : in out Set;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
```
33/2

{*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. If an element equivalent to New_Item is already present in Container at a position other than Position, Program_Error is propagated. Otherwise, Replace_Element assigns New_Item to the element designated by Position. Any exception raised by the assignment is propagated. 34/2

Implementation Note: The final assignment may require that the node of the element be moved in the Set's data structures. That could mean that implementing this operation exactly as worded above could require the overhead of searching twice. Implementations are encouraged to avoid this extra overhead when possible, by prechecking if the old element is equivalent to the new one, by inserting a placeholder node while checking for an equivalent element, and similar optimizations. 34.a/2

The cursor still designates the same element after this operation; only the value of that element has changed. Cursors cannot include information about the relative position of an element in a Set (as they must survive insertions and deletions of other elements), so this should not pose an implementation hardship. 34.b/2

```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```
35/2

{*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated. 36/2

37/2
```
procedure Move (Target : in out Set;
                Source : in out Set);
```

38/2 {*AI95-00302-03*} If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first clears Target. Then, each element from Source is removed from Source and inserted into Target. The length of Source is 0 after a successful call to Move.

39/2
```
procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

40/2 {*AI95-00302-03*} Insert checks if an element equivalent to New_Item is already present in Container. If a match is found, Inserted is set to False and Position designates the matching element. Otherwise, Insert adds New_Item to Container; Inserted is set to True and Position designates the newly-inserted element. Any exception raised during allocation is propagated and Container is not modified.

41/2
```
procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type);
```

42/2 {*AI95-00302-03*} Insert inserts New_Item into Container as per the four-parameter Insert, with the difference that if an element equivalent to New_Item is already in the set, then Constraint_Error is propagated.

42.a/2 **Discussion:** This is equivalent to:

42.b/2
```
declare
  Inserted : Boolean; C : Cursor;
begin
  Insert (Container, New_Item, C, Inserted);
  if not Inserted then
     raise Constraint_Error;
  end if;
end;
```

42.c/2 but doesn't require the hassle of **out** parameters.

43/2
```
procedure Include (Container : in out Set;
                   New_Item  : in     Element_Type);
```

44/2 {*AI95-00302-03*} Include inserts New_Item into Container as per the four-parameter Insert, with the difference that if an element equivalent to New_Item is already in the set, then it is replaced. Any exception raised during assignment is propagated.

45/2
```
procedure Replace (Container : in out Set;
                   New_Item  : in     Element_Type);
```

46/2 {*AI95-00302-03*} Replace checks if an element equivalent to New_Item is already in the set. If a match is found, that element is replaced with New_Item; otherwise, Constraint_Error is propagated.

47/2
```
procedure Exclude (Container : in out Set;
                   Item      : in     Element_Type);
```

48/2 {*AI95-00302-03*} Exclude checks if an element equivalent to Item is present in Container. If a match is found, Exclude removes the element from the set.

49/2
```
procedure Delete (Container : in out Set;
                  Item      : in     Element_Type);
```

50/2 {*AI95-00302-03*} Delete checks if an element equivalent to Item is present in Container. If a match is found, Delete removes the element from the set; otherwise, Constraint_Error is propagated.

51/2
```
procedure Delete (Container : in out Set;
                  Position  : in out Cursor);
```

52/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete removes the element designated by Position from the set. Position is set to No_Element on return.

52.a/2 **Ramification:** The check on Position checks that the cursor does not belong to some other set. This check implies that a reference to the set is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception).

53/2
```
procedure Union (Target : in out Set;
                 Source : in     Set);
```

54/2 {*AI95-00302-03*} Union inserts into Target the elements of Source that are not equivalent to some element already in Target.

54.a/2 **Implementation Note:** If the objects are the same, the result is the same as the original object. The implementation needs to take care so that aliasing effects do not make the result trash; Union (S, S); must work.

55/2
```
function Union (Left, Right : Set) return Set;
```

56/2 {*AI95-00302-03*} Returns a set comprising all of the elements of Left, and the elements of Right that are not equivalent to some element of Left.

57/2
```
procedure Intersection (Target : in out Set;
                        Source : in     Set);
```

58/2 {*AI95-00302-03*} Union deletes from Target the elements of Target that are not equivalent to some element of Source.

58.a/2 **Implementation Note:** If the objects are the same, the result is the same as the original object. The implementation needs to take care so that aliasing effects do not make the result trash; Intersection (S, S); must work.

59/2
```
function Intersection (Left, Right : Set) return Set;
```

60/2 {*AI95-00302-03*} Returns a set comprising all the elements of Left that are equivalent to the some element of Right.

61/2
```
procedure Difference (Target : in out Set;
                      Source : in     Set);
```

62/2 {*AI95-00302-03*} If Target denotes the same object as Source, then Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source.

63/2
```
function Difference (Left, Right : Set) return Set;
```

64/2 {*AI95-00302-03*} Returns a set comprising the elements of Left that are not equivalent to some element of Right.

65/2
```
procedure Symmetric_Difference (Target : in out Set;
                                Source : in     Set);
```

66/2 {*AI95-00302-03*} If Target denotes the same object as Source, then Symmetric_Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source, and inserts into Target the elements of Source that are not equivalent to some element of Target.

67/2
```
function Symmetric_Difference (Left, Right : Set) return Set;
```

68/2 {*AI95-00302-03*} Returns a set comprising the elements of Left that are not equivalent to some element of Right, and the elements of Right that are not equivalent to some element of Left.

69/2
```
function Overlap (Left, Right : Set) return Boolean;
```

70/2 {*AI95-00302-03*} If an element of Left is equivalent to some element of Right, then Overlap returns True. Otherwise it returns False.

70.a/2 **Discussion:** This operation is commutative. If Overlap returns False, the two sets are disjoint.

71/2
```
function Is_Subset (Subset : Set;
                    Of_Set : Set) return Boolean;
```

72/2 {*AI95-00302-03*} If an element of Subset is not equivalent to some element of Of_Set, then Is_Subset returns False. Otherwise it returns True.

72.a/2 **Discussion:** This operation is not commutative, so we use parameter names that make it clear in named notation which set is which.

73/2
```
function First (Container : Set) return Cursor;
```

74/2 {*AI95-00302-03*} If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first element in Container.

75/2
```
function Next (Position  : Cursor) return Cursor;
```

76/2 {*AI95-00302-03*} Returns a cursor that designates the successor of the element designated by Position. If Position designates the last element, then No_Element is returned. If Position equals No_Element, then No_Element is returned.

77/2
```
procedure Next (Position  : in out Cursor);
```

78/2 {*AI95-00302-03*} Equivalent to Position := Next (Position).

79/2 {*AI95-00302-03*} Equivalent to Find (Container, Item) /= No_Element.

80/2
```
function Find (Container : Set;
               Item      : Element_Type) return Cursor;
```

81/2 {*AI95-00302-03*} If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if an element equivalent to Item is present in Container. If a match is found, a cursor designating the matching element is returned; otherwise, No_Element is returned.

82/2
```
function Contains (Container : Set;
                   Item      : Element_Type) return Boolean;
```

83/2
```
function Has_Element (Position : Cursor) return Boolean;
```

84/2 {*AI95-00302-03*} Returns True if Position designates an element, and returns False otherwise.

84.a/2 **To be honest:** This function may not detect cursors that designate deleted elements; such cursors are invalid (see below); the result of Has_Element for invalid cursors is unspecified (but not erroneous).

85/2
```
procedure Iterate
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor));
```

86/2 {*AI95-00302-03*} Iterate calls Process.**all** with a cursor that designates each element in Container, starting with the first element and moving the cursor according to the successor relation. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

86.a/2 **Implementation Note:** The “tamper with cursors” check takes place when the operations that insert or delete elements, and so on are called.

86.b/2 See Iterate for vectors (A.18.2) for a suggested implementation of the check.

87/2 {*AI95-00302-03*} Both Containers.Hashed_Set and Containers.Ordered_Set declare a nested generic package Generic_Keys, which provides operations that allow set manipulation in terms of a key (typically, a portion of an element) instead of a complete element. The formal function Key of

Generic_Keys extracts a key value from an element. It is expected to return the same value each time it is called with a particular element. The behavior of Generic_Keys is unspecified if Key behaves in some other manner.{*unspecified* [partial]}

88/2 {*AI95-00302-03*} A key is expected to unambiguously determine a single equivalence class for elements. The behavior of Generic_Keys is unspecified if the formal parameters of this package behave in some other manner.{*unspecified* [partial]}

89/2
```
function Key (Position : Cursor) return Key_Type;
```

90/2 {*AI95-00302-03*} Equivalent to Key (Element (Position)).

91/2 {*AI95-00302-03*} The subprograms in package Generic_Keys named Contains, Find, Element, Delete, and Exclude, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key parameter is used to locate an element in the set.

92/2
```
procedure Replace (Container : in out Set;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);
```

93/2 {*AI95-00302-03*} Equivalent to Replace_Element (Container, Find (Container, Key), New_Item).

94/2
```
procedure Update_Element_Preserving_Key
  (Container : in out Set;
   Position  : in     Cursor;
   Process   : not null access procedure
                              (Element : in out Element_Type));
```

95/2 {*AI95-00302-03*} If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Update_Element_Preserving_Key uses Key to save the key value *K* of the element designated by Position. Update_Element_Preserving_Key then calls Process.**all** with that element as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated. After Process.**all** returns, Update_Element_-Preserving_Key checks if *K* determines the same equivalence class as that for the new element; if not, the element is removed from the set and Program_Error is propagated.

95.a/2 **Reason:** The key check ensures that the invariants of the set are preserved by the modification. The “tampers with the elements” check prevents data loss (if Element_Type is by-copy) or erroneous execution (if element type is unconstrained and indefinite).

96/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

96.a/2 **Ramification:** This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place.

Erroneous Execution

97/2 {*AI95-00302-03*} A Cursor value is *invalid* if any of the following have occurred since it was created:{*invalid cursor (of a set)*} {*cursor (invalid)* [partial]}

- 98/2 The set that contains the element it designates has been finalized;
- 99/2 The set that contains the element it designates has been used as the Source or Target of a call to Move; or
- 100/2 The element it designates has been deleted from the set.

101/2 {*AI95-00302-03*} The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter.{*unspecified* [partial]} Execution is erroneous if any other subprogram declared in

Containers.Hashed_Sets or Containers.Ordered_Sets is called with an invalid cursor parameter.{*erroneous execution (cause)* [partial]}

101.a/2 **Discussion:** The list above is intended to be exhaustive. In other cases, a cursor value continues to designate its original element. For instance, cursor values survive the insertion and deletion of other elements.

101.b/2 While it is possible to check for these cases, in many cases the overhead necessary to make the check is substantial in time or space. Implementations are encouraged to check for as many of these cases as possible and raise Program_Error if detected.

Implementation Requirements

102/2 {*AI95-00302-03*} No storage associated with a Set object shall be lost upon assignment or scope exit.

103/2 {*AI95-00302-03*} The execution of an assignment_statement for a set shall have the effect of copying the elements from the source set object to the target set object.

103.a/2 **Implementation Note:** An assignment of a Set is a “deep” copy; that is the elements are copied as well as the data structures. We say “effect of” in order to allow the implementation to avoid copying elements immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed.

Implementation Advice

104/2 {*AI95-00302-03*} Move should not copy elements, and should minimize copying of internal data structures.

104.a/2 **Implementation Advice:** Move for sets should not copy elements, and should minimize copying of internal data structures.

104.b/2 **Implementation Note:** Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source container to the Target container.

105/2 {*AI95-00302-03*} If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation.

105.a/2 **Implementation Advice:** If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation.

105.b/2 **Reason:** This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished.

Wording Changes from Ada 95

105.c/2 {*AI95-00302-03*} This description of sets is new; the extensions are documented with the specific packages.

A.18.8 The Package Containers.Hashed_Sets

Static Semantics

1/2 {*AI95-00302-03*} The generic library package Containers.Hashed_Sets has the following declaration:

```
2/2 generic
       type Element_Type is private;
       with function Hash (Element : Element_Type) return Hash_Type;
       with function Equivalent_Elements (Left, Right : Element_Type)
                     return Boolean;
       with function "=" (Left, Right : Element_Type) return Boolean is <>;
    package Ada.Containers.Hashed_Sets is
       pragma Preelaborate(Hashed_Sets);

3/2    type Set is tagged private;
       pragma Preelaborable_Initialization(Set);

4/2    type Cursor is private;
       pragma Preelaborable_Initialization(Cursor);

5/2    Empty_Set : constant Set;

6/2    No_Element : constant Cursor;
```

```
function "=" (Left, Right : Set) return Boolean;                          7/2

function Equivalent_Sets (Left, Right : Set) return Boolean;              8/2

function To_Set (New_Item : Element_Type) return Set;                     9/2

function Capacity (Container : Set) return Count_Type;                    10/2

procedure Reserve_Capacity (Container : in out Set;                       11/2
                            Capacity  : in     Count_Type);

function Length (Container : Set) return Count_Type;                      12/2

function Is_Empty (Container : Set) return Boolean;                       13/2

procedure Clear (Container : in out Set);                                 14/2

function Element (Position : Cursor) return Element_Type;                 15/2

procedure Replace_Element (Container : in out Set;                        16/2
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);

procedure Query_Element                                                   17/2
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));

procedure Move (Target : in out Set;                                      18/2
                Source : in out Set);

procedure Insert (Container : in out Set;                                 19/2
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);

procedure Insert (Container : in out Set;                                 20/2
                  New_Item  : in     Element_Type);

procedure Include (Container : in out Set;                                21/2
                   New_Item  : in     Element_Type);

procedure Replace (Container : in out Set;                                22/2
                   New_Item  : in     Element_Type);

procedure Exclude (Container : in out Set;                                23/2
                   Item      : in     Element_Type);

procedure Delete (Container : in out Set;                                 24/2
                  Item      : in     Element_Type);

procedure Delete (Container : in out Set;                                 25/2
                  Position  : in out Cursor);

procedure Union (Target : in out Set;                                     26/2
                 Source : in     Set);

function Union (Left, Right : Set) return Set;                            27/2

function "or" (Left, Right : Set) return Set renames Union;               28/2

procedure Intersection (Target : in out Set;                              29/2
                        Source : in     Set);

function Intersection (Left, Right : Set) return Set;                     30/2

function "and" (Left, Right : Set) return Set renames Intersection;       31/2

procedure Difference (Target : in out Set;                                32/2
                      Source : in     Set);

function Difference (Left, Right : Set) return Set;                       33/2

function "-" (Left, Right : Set) return Set renames Difference;           34/2

procedure Symmetric_Difference (Target : in out Set;                      35/2
                                Source : in     Set);

function Symmetric_Difference (Left, Right : Set) return Set;             36/2

function "xor" (Left, Right : Set) return Set                             37/2
  renames Symmetric_Difference;

function Overlap (Left, Right : Set) return Boolean;                      38/2
```

```
39/2       function Is_Subset (Subset : Set;
                               Of_Set : Set) return Boolean;
40/2       function First (Container : Set) return Cursor;
41/2       function Next (Position : Cursor) return Cursor;
42/2       procedure Next (Position : in out Cursor);
43/2       function Find (Container : Set;
                          Item      : Element_Type) return Cursor;
44/2       function Contains (Container : Set;
                              Item      : Element_Type) return Boolean;
45/2       function Has_Element (Position : Cursor) return Boolean;
46/2       function Equivalent_Elements (Left, Right : Cursor)
             return Boolean;
47/2       function Equivalent_Elements (Left  : Cursor;
                                         Right : Element_Type)
             return Boolean;
48/2       function Equivalent_Elements (Left  : Element_Type;
                                         Right : Cursor)
             return Boolean;
49/2       procedure Iterate
             (Container : in Set;
              Process   : not null access procedure (Position : in Cursor));
50/2       generic
              type Key_Type (<>) is private;
              with function Key (Element : Element_Type) return Key_Type;
              with function Hash (Key : Key_Type) return Hash_Type;
              with function Equivalent_Keys (Left, Right : Key_Type)
                                             return Boolean;
           package Generic_Keys is
51/2          function Key (Position : Cursor) return Key_Type;
52/2          function Element (Container : Set;
                                Key       : Key_Type)
                return Element_Type;
53/2          procedure Replace (Container : in out Set;
                                 Key       : in     Key_Type;
                                 New_Item  : in     Element_Type);
54/2          procedure Exclude (Container : in out Set;
                                 Key       : in     Key_Type);
55/2          procedure Delete (Container : in out Set;
                                Key       : in     Key_Type);
56/2          function Find (Container : Set;
                             Key       : Key_Type)
                 return Cursor;
57/2          function Contains (Container : Set;
                                 Key       : Key_Type)
                 return Boolean;
58/2          procedure Update_Element_Preserving_Key
                (Container : in out Set;
                 Position  : in     Cursor;
                 Process   : not null access procedure
                                 (Element : in out Element_Type));
59/2       end Generic_Keys;
60/2    private
61/2       ... -- not specified by the language
62/2    end Ada.Containers.Hashed_Sets;
```

{*AI95-00302-03*} {*capacity (of a hashed set)*} An object of type Set contains an expandable hash table, which is used to provide direct access to elements. The *capacity* of an object of type Set is the maximum number of elements that can be inserted into the hash table prior to it being automatically expanded. 63/2

{*AI95-00302-03*} {*equivalent element (of a hashed set)*} Two elements *E1* and *E2* are defined to be *equivalent* if Equivalent_Elements (*E1*, *E2*) returns True. 64/2

{*AI95-00302-03*} The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular element value. For any two equivalent elements, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.{*unspecified* [partial]} 65/2

{*AI95-00302-03*} The actual function for the generic formal function Equivalent_Elements is expected to return the same value each time it is called with a particular pair of Element values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Elements behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Elements, and how many times they call it, is unspecified.{*unspecified* [partial]} 66/2

{*AI95-00302-03*} If the value of an element stored in a set is changed other than by an operation in this package such that at least one of Hash or Equivalent_Elements give different results, the behavior of this package is unspecified.{*unspecified* [partial]} 67/2

> **Discussion:** See A.18.5, “The Package Containers.Hashed_Maps” for a suggested implementation, and for justification of the restrictions regarding Hash and Equivalent_Elements. Note that sets only need to store elements, not key/element pairs. 67.a/2

{*AI95-00302-03*} {*first element (of a hashed set)*} {*last element (of a hashed set)*} {*successor element (of a hashed set)*} Which elements are the first element and the last element of a set, and which element is the successor of a given element, are unspecified, other than the general semantics described in A.18.7.{*unspecified* [partial]} 68/2

```
function Capacity (Container : Set) return Count_Type;
```
69/2

{*AI95-00302-03*} Returns the capacity of Container. 70/2

```
procedure Reserve_Capacity (Container : in out Set;
                            Capacity  : in     Count_Type);
```
71/2

{*AI95-00302-03*} Reserve_Capacity allocates a new hash table such that the length of the resulting set can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the elements in Container onto the new hash table. It replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified. 72/2

Reserve_Capacity tampers with the cursors of Container. 73/2

> **Reason:** Reserve_Capacity tampers with the cursors, as rehashing probably will change the relationships of the elements in Container. 73.a/2

```
procedure Clear (Container : in out Set);
```
74/2

{*AI95-00302-03*} In addition to the semantics described in A.18.7, Clear does not affect the capacity of Container. 75/2

76/2
```
procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

77/2 {*AI95-00302-03*} In addition to the semantics described in A.18.7, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value.

78/2
```
function First (Container : Set) return Cursor;
```

79/2 {*AI95-00302-03*} If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first hashed element in Container.

80/2
```
function Equivalent_Elements (Left, Right : Cursor)
      return Boolean;
```

81/2 {*AI95-00302-03*} Equivalent to Equivalent_Elements (Element (Left), Element (Right)).

82/2
```
function Equivalent_Elements (Left  : Cursor;
                              Right : Element_Type) return Boolean;
```

83/2 {*AI95-00302-03*} Equivalent to Equivalent_Elements (Element (Left), Right).

84/2
```
function Equivalent_Elements (Left  : Element_Type;
                              Right : Cursor) return Boolean;
```

85/2 {*AI95-00302-03*} Equivalent to Equivalent_Elements (Left, Element (Right)).

86/2 {*AI95-00302-03*} For any element *E*, the actual function for the generic formal function Generic_Keys.Hash is expected to be such that Hash (*E*) = Generic_Keys.Hash (Key (*E*)). If the actuals for Key or Generic_Keys.Hash behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Generic_Keys.Hash, and how many times they call it, is unspecified.{*unspecified* [partial]}

87/2 {*AI95-00302-03*} For any two elements *E1* and *E2*, the boolean values Equivalent_Elements (*E1*, *E2*) and Equivalent_Keys (Key (*E1*), Key (*E2*)) are expected to be equal. If the actuals for Key or Equivalent_Keys behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Equivalent_Keys, and how many times they call it, is unspecified.{*unspecified* [partial]}

Implementation Advice

88/2 {*AI95-00302-03*} If *N* is the length of a set, the average time complexity of the subprograms Insert, Include, Replace, Delete, Exclude and Find that take an element parameter should be *O*(log *N*). The average time complexity of the subprograms that take a cursor parameter should be *O*(1). The average time complexity of Reserve_Capacity should be *O*(*N*).

88.a/2 **Implementation Advice:** The average time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations of Containers.Hashed_Sets that take an element parameter should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Sets that take a cursor parameter should be *O*(1). The average time complexity of Containers.Hashed_Sets.Reserve_Capacity should be *O*(*N*).

88.b/2 **Implementation Note:** {*AI95-00302-03*} See A.18.5, “The Package Containers.Hashed_Maps” for implementation notes regarding some of the operations of this package.

Extensions to Ada 95

88.c/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Hashed_Sets is new.

A.18.9 The Package Containers.Ordered_Sets

Static Semantics

{*AI95-00302-03*} The generic library package Containers.Ordered_Sets has the following declaration: 1/2

```
generic                                                                              2/2
   type Element_Type is private;
   with function "<" (Left, Right : Element_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets is
   pragma Preelaborate(Ordered_Sets);

   function Equivalent_Elements (Left, Right : Element_Type) return Boolean;        3/2

   type Set is tagged private;                                                       4/2
   pragma Preelaborable_Initialization(Set);

   type Cursor is private;                                                           5/2
   pragma Preelaborable_Initialization(Cursor);

   Empty_Set : constant Set;                                                         6/2

   No_Element : constant Cursor;                                                     7/2

   function "=" (Left, Right : Set) return Boolean;                                  8/2

   function Equivalent_Sets (Left, Right : Set) return Boolean;                      9/2

   function To_Set (New_Item : Element_Type) return Set;                            10/2

   function Length (Container : Set) return Count_Type;                             11/2

   function Is_Empty (Container : Set) return Boolean;                              12/2

   procedure Clear (Container : in out Set);                                        13/2

   function Element (Position : Cursor) return Element_Type;                        14/2

   procedure Replace_Element (Container : in out Set;                               15/2
                              Position  : in     Cursor;
                              New_Item  : in     Element_Type);

   procedure Query_Element                                                          16/2
     (Position : in Cursor;
      Process  : not null access procedure (Element : in Element_Type));

   procedure Move (Target : in out Set;                                             17/2
                   Source : in out Set);

   procedure Insert (Container : in out Set;                                        18/2
                     New_Item  : in     Element_Type;
                     Position  :    out Cursor;
                     Inserted  :    out Boolean);

   procedure Insert (Container : in out Set;                                        19/2
                     New_Item  : in     Element_Type);

   procedure Include (Container : in out Set;                                       20/2
                      New_Item  : in     Element_Type);

   procedure Replace (Container : in out Set;                                       21/2
                      New_Item  : in     Element_Type);

   procedure Exclude (Container : in out Set;                                       22/2
                      Item      : in     Element_Type);

   procedure Delete (Container : in out Set;                                        23/2
                     Item      : in     Element_Type);

   procedure Delete (Container : in out Set;                                        24/2
                     Position  : in out Cursor);

   procedure Delete_First (Container : in out Set);                                 25/2

   procedure Delete_Last (Container : in out Set);                                  26/2
```

```
27/2    procedure Union (Target : in out Set;
                         Source : in     Set);
28/2    function Union (Left, Right : Set) return Set;
29/2    function "or" (Left, Right : Set) return Set renames Union;
30/2    procedure Intersection (Target : in out Set;
                                Source : in     Set);
31/2    function Intersection (Left, Right : Set) return Set;
32/2    function "and" (Left, Right : Set) return Set renames Intersection;
33/2    procedure Difference (Target : in out Set;
                              Source : in     Set);
34/2    function Difference (Left, Right : Set) return Set;
35/2    function "-" (Left, Right : Set) return Set renames Difference;
36/2    procedure Symmetric_Difference (Target : in out Set;
                                        Source : in     Set);
37/2    function Symmetric_Difference (Left, Right : Set) return Set;
38/2    function "xor" (Left, Right : Set) return Set renames
           Symmetric_Difference;
39/2    function Overlap (Left, Right : Set) return Boolean;
40/2    function Is_Subset (Subset : Set;
                            Of_Set : Set) return Boolean;
41/2    function First (Container : Set) return Cursor;
42/2    function First_Element (Container : Set) return Element_Type;
43/2    function Last (Container : Set) return Cursor;
44/2    function Last_Element (Container : Set) return Element_Type;
45/2    function Next (Position : Cursor) return Cursor;
46/2    procedure Next (Position : in out Cursor);
47/2    function Previous (Position : Cursor) return Cursor;
48/2    procedure Previous (Position : in out Cursor);
49/2    function Find (Container : Set;
                       Item      : Element_Type)
           return Cursor;
50/2    function Floor (Container : Set;
                        Item      : Element_Type)
           return Cursor;
51/2    function Ceiling (Container : Set;
                          Item      : Element_Type)
           return Cursor;
52/2    function Contains (Container : Set;
                           Item      : Element_Type) return Boolean;
53/2    function Has_Element (Position : Cursor) return Boolean;
54/2    function "<" (Left, Right : Cursor) return Boolean;
55/2    function ">" (Left, Right : Cursor) return Boolean;
56/2    function "<" (Left : Cursor; Right : Element_Type)
           return Boolean;
57/2    function ">" (Left : Cursor; Right : Element_Type)
           return Boolean;
58/2    function "<" (Left : Element_Type; Right : Cursor)
           return Boolean;
59/2    function ">" (Left : Element_Type; Right : Cursor)
           return Boolean;
```

```
   procedure Iterate                                                          60/2
     (Container : in Set;
      Process   : not null access procedure (Position : in Cursor));

   procedure Reverse_Iterate                                                  61/2
     (Container : in Set;
      Process   : not null access procedure (Position : in Cursor));

   generic                                                                    62/2
      type Key_Type (<>) is private;
      with function Key (Element : Element_Type) return Key_Type;
      with function "<" (Left, Right : Key_Type)
         return Boolean is <>;
   package Generic_Keys is

       function Equivalent_Keys (Left, Right : Key_Type)                      63/2
          return Boolean;

       function Key (Position : Cursor) return Key_Type;                      64/2

       function Element (Container : Set;                                     65/2
                         Key       : Key_Type)
          return Element_Type;

       procedure Replace (Container : in out Set;                             66/2
                          Key       : in     Key_Type;
                          New_Item  : in     Element_Type);

       procedure Exclude (Container : in out Set;                             67/2
                          Key       : in     Key_Type);

       procedure Delete (Container : in out Set;                              68/2
                         Key       : in     Key_Type);

       function Find (Container : Set;                                        69/2
                      Key       : Key_Type)
          return Cursor;

       function Floor (Container : Set;                                       70/2
                       Key       : Key_Type)
          return Cursor;

       function Ceiling (Container : Set;                                     71/2
                         Key       : Key_Type)
          return Cursor;

       function Contains (Container : Set;                                    72/2
                          Key       : Key_Type) return Boolean;

       procedure Update_Element_Preserving_Key                                73/2
         (Container : in out Set;
          Position  : in     Cursor;
          Process   : not null access procedure
                          (Element : in out Element_Type));

   end Generic_Keys;                                                          74/2

private                                                                       75/2

   ... -- not specified by the language                                       76/2

end Ada.Containers.Ordered_Sets;                                              77/2
```

{*AI95-00302-03*} Two elements *E1* and *E2* are *equivalent* if both *E1* < *E2* and *E2* < *E1* return False, using the generic formal "<" operator for elements.{*equivalent element (of a ordered set)*} Function Equivalent_Elements returns True if Left and Right are equivalent, and False otherwise. 78/2

{*AI95-00302-03*} The actual function for the generic formal function "<" on Element_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive. If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.{*unspecified* [partial]} 79/2

80/2 {*AI95-00302-03*} If the value of an element stored in a set is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.{*unspecified* [partial]}

80.a/2 **Discussion:** See A.18.6, “The Package Containers.Ordered_Maps” for a suggested implementation, and for justification of the restrictions regarding "<" and "=". Note that sets only need to store elements, not key/element pairs.

81/2 {*AI95-00302-03*} {*first element (of a ordered set)*} {*last element (of a ordered set)*} {*successor element (of a ordered set)*} The first element of a nonempty set is the one which is less than all the other elements in the set. The last element of a nonempty set is the one which is greater than all the other elements in the set. The successor of an element is the smallest element that is larger than the given element. The predecessor of an element is the largest element that is smaller than the given element. All comparisons are done using the generic formal "<" operator for elements.

82/2
```
procedure Delete_First (Container : in out Set);
```

83/2 {*AI95-00302-03*} If Container is empty, Delete_First has no effect. Otherwise the element designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container.

84/2
```
procedure Delete_Last (Container : in out Set);
```

85/2 {*AI95-00302-03*} If Container is empty, Delete_Last has no effect. Otherwise the element designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container.

86/2
```
function First_Element (Container : Set) return Element_Type;
```

87/2 {*AI95-00302-03*} Equivalent to Element (First (Container)).

88/2
```
function Last (Container : Set) return Cursor;
```

89/2 {*AI95-00302-03*} Returns a cursor that designates the last element in Container. If Container is empty, returns No_Element.

90/2
```
function Last_Element (Container : Set) return Element_Type;
```

91/2 {*AI95-00302-03*} Equivalent to Element (Last (Container)).

92/2
```
function Previous (Position : Cursor) return Cursor;
```

93/2 {*AI95-00302-03*} If Position equals No_Element, then Previous returns No_Element. Otherwise Previous returns a cursor designating the element that precedes the one designated by Position. If Position designates the first element, then Previous returns No_Element.

94/2
```
procedure Previous (Position : in out Cursor);
```

95/2 {*AI95-00302-03*} Equivalent to Position := Previous (Position).

96/2
```
function Floor (Container : Set;
                Item      : Element_Type) return Cursor;
```

97/2 {*AI95-00302-03*} Floor searches for the last element which is not greater than Item. If such an element is found, a cursor that designates it is returned. Otherwise No_Element is returned.

98/2
```
function Ceiling (Container : Set;
                  Item      : Element_Type) return Cursor;
```

99/2 {*AI95-00302-03*} Ceiling searches for the first element which is not less than Item. If such an element is found, a cursor that designates it is returned. Otherwise No_Element is returned.

100/2
```
function "<" (Left, Right : Cursor) return Boolean;
```

101/2 {*AI95-00302-03*} Equivalent to Element (Left) < Element (Right).

```
function ">" (Left, Right : Cursor) return Boolean;
```
102/2

{*AI95-00302-03*} Equivalent to Element (Right) < Element (Left). 103/2

```
function "<" (Left : Cursor; Right : Element_Type) return Boolean;
```
104/2

{*AI95-00302-03*} Equivalent to Element (Left) < Right. 105/2

```
function ">" (Left : Cursor; Right : Element_Type) return Boolean;
```
106/2

{*AI95-00302-03*} Equivalent to Right < Element (Left). 107/2

```
function "<" (Left : Element_Type; Right : Cursor) return Boolean;
```
108/2

{*AI95-00302-03*} Equivalent to Left < Element (Right). 109/2

```
function ">" (Left : Element_Type; Right : Cursor) return Boolean;
```
110/2

{*AI95-00302-03*} Equivalent to Element (Right) < Left. 111/2

```
procedure Reverse_Iterate
   (Container : in Set;
    Process   : not null access procedure (Position : in Cursor));
```
112/2

{*AI95-00302-03*} Iterates over the elements in Container as per Iterate, with the difference that the elements are traversed in predecessor order, starting with the last element. 113/2

{*AI95-00302-03*} For any two elements *E1* and *E2*, the boolean values (*E1* < *E2*) and (Key(*E1*) < Key(*E2*)) are expected to be equal. If the actuals for Key or Generic_Keys."<" behave in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Key and Generic_Keys."<", and how many times the functions are called, is unspecified.{*unspecified* [partial]} 114/2

{*AI95-00302-03*} In addition to the semantics described in A.18.7, the subprograms in package Generic_Keys named Floor and Ceiling, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key subprogram parameter is compared to elements in the container using the Key and "<" generic formal functions. The function named Equivalent_Keys in package Generic_Keys returns True if both Left < Right and Right < Left return False using the generic formal "<" operator, and returns True otherwise. 115/2

Implementation Advice

{*AI95-00302-03*} If *N* is the length of a set, then the worst-case time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations that take an element parameter should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms that take a cursor parameter should be *O*(1). 116/2

Implementation Advice: The worst-case time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations of Containers.Ordered_Sets that take an element parameter should be *O*((log *N*)**2). The worst-case time complexity of the subprograms of Containers.Ordered_Sets that take a cursor parameter should be *O*(1). 116.a/2

Implementation Note: {*AI95-00302-03*} See A.18.6, "The Package Containers.Ordered_Maps" for implementation notes regarding some of the operations of this package. 116.b/2

Extensions to Ada 95

{*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Ordered_Sets is new. 116.c/2

A.18.10 The Package Containers.Indefinite_Vectors

{*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Vectors provides a private type Vector and a set of operations. It provides the same operations as the package Containers.Vectors (see A.18.2), with the difference that the generic formal Element_Type is indefinite. 1/2

Static Semantics

2/2 {*AI95-00302-03*} The declaration of the generic library package Containers.Indefinite_Vectors has the same contents as Containers.Vectors except:

3/2
- The generic formal Element_Type is indefinite.

4/2
- The procedures with the profiles:

5/2
```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1);
```

6
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

7/2 are omitted.

7.a/2 **Discussion:** These procedures are omitted because there is no way to create a default-initialized object of an indefinite type. Note that Insert_Space can be used instead of this routine in most cases. Omitting the routine completely allows any problems to be diagnosed by the compiler when converting from a definite to indefinite vector.

8/2
- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

Extensions to Ada 95

8.a/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Indefinite_Vectors is new.

A.18.11 The Package Containers.Indefinite_Doubly_Linked_Lists

1/2 {*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. It provides the same operations as the package Containers.Doubly_Linked_Lists (see A.18.3), with the difference that the generic formal Element_Type is indefinite.

Static Semantics

2/2 {*AI95-00302-03*} The declaration of the generic library package Containers.Indefinite_Doubly_Linked_-Lists has the same contents as Containers.Doubly_Linked_Lists except:

3/2
- The generic formal Element_Type is indefinite.

4/2
- The procedure with the profile:

5/2
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

6/2 is omitted.

6.a/2 **Discussion:** This procedure is omitted because there is no way to create a default-initialized object of an indefinite type. We considered having this routine insert an empty element similar to the empty elements of a vector, but rejected this possibility because the semantics are fairly complex and very different from the existing case. That would make it more error-prone to convert a container from a definite type to an indefinite type; by omitting the routine completely, any problems will be diagnosed by the compiler.

7/2
- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

Extensions to Ada 95

7.a/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Indefinite_Doubly_Linked_Lists is new.

A.18.12 The Package Containers.Indefinite_Hashed_Maps

{*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Hashed_Maps provides a map with the same operations as the package Containers.Hashed_Maps (see A.18.5), with the difference that the generic formal types Key_Type and Element_Type are indefinite. 1/2

Static Semantics

{*AI95-00302-03*} The declaration of the generic library package Containers.Indefinite_Hashed_Maps has the same contents as Containers.Hashed_Maps except: 2/2

- The generic formal Key_Type is indefinite. 3/2
- The generic formal Element_Type is indefinite. 4/2
- The procedure with the profile: 5/2

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```
6/2

is omitted. 7/2

Discussion: This procedure is omitted because there is no way to create a default-initialized object of an indefinite type. We considered having this routine insert an empty element similar to the empty elements of a vector, but rejected this possibility because the semantics are fairly complex and very different from the existing case. That would make it more error-prone to convert a container from a definite type to an indefinite type; by omitting the routine completely, any problems will be diagnosed by the compiler. 7.a/2

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained. 8/2

Extensions to Ada 95

{*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Indefinite_Hashed_Maps is new. 8.a/2

A.18.13 The Package Containers.Indefinite_Ordered_Maps

{*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Ordered_Maps provides a map with the same operations as the package Containers.Ordered_Maps (see A.18.6), with the difference that the generic formal types Key_Type and Element_Type are indefinite. 1/2

Static Semantics

{*AI95-00302-03*} The declaration of the generic library package Containers.Indefinite_Ordered_Maps has the same contents as Containers.Ordered_Maps except: 2/2

- The generic formal Key_Type is indefinite. 3/2
- The generic formal Element_Type is indefinite. 4/2
- The procedure with the profile: 5/2

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```
6/2

is omitted. 7/2

Discussion: This procedure is omitted because there is no way to create a default-initialized object of an indefinite type. We considered having this routine insert an empty element similar to the empty elements of a vector, but rejected this possibility because the semantics are fairly complex and very different from the existing case. That would make it more error-prone to convert a container from a definite type to an indefinite type; by omitting the routine completely, any problems will be diagnosed by the compiler. 7.a/2

8/2 • The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

Extensions to Ada 95

8.a/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Indefinite_Ordered_Maps is new.

A.18.14 The Package Containers.Indefinite_Hashed_Sets

1/2 {*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Hashed_Sets provides a set with the same operations as the package Containers.Hashed_Sets (see A.18.8), with the difference that the generic formal type Element_Type is indefinite.

Static Semantics

2/2 {*AI95-00302-03*} The declaration of the generic library package Containers.Indefinite_Hashed_Sets has the same contents as Containers.Hashed_Sets except:

3/2 • The generic formal Element_Type is indefinite.

4/2 • The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

Extensions to Ada 95

4.a/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Indefinite_Hashed_Sets is new.

A.18.15 The Package Containers.Indefinite_Ordered_Sets

1/2 {*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Ordered_Sets provides a set with the same operations as the package Containers.Ordered_Sets (see A.18.9), with the difference that the generic formal type Element_Type is indefinite.

Static Semantics

2/2 {*AI95-00302-03*} The declaration of the generic library package Containers.Indefinite_Ordered_Sets has the same contents as Containers.Ordered_Sets except:

3/2 • The generic formal Element_Type is indefinite.

4/2 • The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

Extensions to Ada 95

4.a/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic package Containers.Indefinite_Ordered_Sets is new.

A.18.16 Array Sorting

1/2 {*AI95-00302-03*} The language-defined generic procedures Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort provide sorting on arbitrary array types.

Static Semantics

{*AI95-00302-03*} The generic library procedure Containers.Generic_Array_Sort has the following declaration: 2/2

3/2
```
generic
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type range <>) of Element_Type;
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
procedure Ada.Containers.Generic_Array_Sort (Container : in out Array_Type);
pragma Pure(Ada.Containers.Generic_Array_Sort);
```

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated. 4/2

The actual function for the generic formal function "<" of Generic_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Array_Sort is unspecified. How many times Generic_Array_Sort calls "<" is unspecified.{*unspecified* [partial]} 5/2

Ramification: This implies swapping the elements, usually including an intermediate copy. This of course means that the elements will be copied. Since the elements are nonlimited, this usually will not be a problem. Note that there is Implementation Advice below that the implementation should use a sort that minimizes copying of elements. 5.a/2

The sort is not required to be stable (and the fast algorithm required will not be stable). If a stable sort is needed, the user can include the original location of the element as an extra "sort key". We considered requiring the implementation to do that, but it is mostly extra overhead -- usually there is something already in the element that provides the needed stability. 5.b/2

{*AI95-00302-03*} The generic library procedure Containers.Generic_Constrained_Array_Sort has the following declaration: 6/2

7/2
```
generic
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type) of Element_Type;
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
procedure Ada.Containers.Generic_Constrained_Array_Sort
      (Container : in out Array_Type);
pragma Pure(Ada.Containers.Generic_Constrained_Array_Sort);
```

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated. 8/2

The actual function for the generic formal function "<" of Generic_Constrained_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Constrained_Array_Sort is unspecified. How many times Generic_Constrained_Array_Sort calls "<" is unspecified.{*unspecified* [partial]} 9/2

Implementation Advice

{*AI95-00302-03*} The worst-case time complexity of a call on an instance of Containers.Generic_Array_Sort or Containers.Generic_Constrained_Array_Sort should be *O*(*N***2) or 10/2 better, and the average time complexity should be better than *O*(*N***2), where *N* is the length of the Container parameter.

10.a/2 **Implementation Advice:** Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2).

10.b/2 **Discussion:** In other words, we're requiring the use of a sorting algorithm better than *O*(*N***2), such as Quicksort. No bubble sorts allowed!

11/2 {*AI95-00302-03*} Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements.

11.a/2 **Implementation Advice:** Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements.

11.b/2 **To be honest:** We do not mean “absolutely minimize” here; we're not intending to require a single copy for each element. Rather, we want to suggest that the sorting algorithm chosen is one that does not copy items unnecessarily. Bubble sort would not meet this advice, for instance.

Extensions to Ada 95

11.c/2 {*AI95-00302-03*} {*extensions to Ada 95*} The generic packages Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort are new.

Annex B
(normative)
Interface to Other Languages

{*interface to other languages*} {*language (interface to non-Ada)*} {*mixed-language programs*} This Annex describes features for writing mixed-language programs. General interface support is presented first; then specific support for C, COBOL, and Fortran is defined, in terms of language interface packages for each of these languages. 1

Ramification: This Annex is not a "Specialized Needs" annex. Every implementation must support all non-optional features defined here (mainly the package Interfaces). 1.a

Language Design Principles

Ada should have strong support for mixed-language programming. 1.b

Extensions to Ada 83

{*extensions to Ada 83*} Much of the functionality in this Annex is new to Ada 95. 1.c

Wording Changes from Ada 83

This Annex contains what used to be RM83-13.8. 1.d

B.1 Interfacing Pragmas

A pragma Import is used to import an entity defined in a foreign language into an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada. In contrast, a pragma Export is used to export an Ada entity to a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language. The pragmas Import and Export are intended primarily for objects and subprograms, although implementations are allowed to support other entities. 1

A pragma Convention is used to specify that an Ada entity should use the conventions of another language. It is intended primarily for types and "callback" subprograms. For example, "**pragma** Convention(Fortran, Matrix);" implies that Matrix should be represented according to the conventions of the supported Fortran implementation, namely column-major order. 2

A pragma Linker_Options is used to specify the system linker parameters needed when a given compilation unit is included in a partition. 3

Syntax

{*interfacing pragma* [distributed]} {*interfacing pragma (Import)* [partial]} {*pragma, interfacing (Import)* [partial]} {*interfacing pragma (Export)* [partial]} {*pragma, interfacing (Export)* [partial]} {*interfacing pragma (Convention)* [partial]} {*pragma, interfacing (Convention)* [partial]} {*pragma, interfacing (Linker_Options)* [partial]} An *interfacing pragma* is a representation pragma that is one of the pragmas Import, Export, or Convention. Their forms, together with that of the related pragma Linker_Options, are as follows: 4

5
pragma Import(
  [Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *string*_expression] [, [Link_Name =>] *string*_expression]);

6
pragma Export(
  [Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *string*_expression] [, [Link_Name =>] *string*_expression]);

7 **pragma** Convention([Convention =>] *convention*_identifier,[Entity =>] local_name);

8 **pragma** Linker_Options(*string*_expression);

9 A pragma Linker_Options is allowed only at the place of a declarative_item.

9.1/1 {*8652/0058*} {*AI95-00036-01*} For pragmas Import and Export, the argument for Link_Name shall not be given without the *pragma_argument*_identifier unless the argument for External_Name is given.

Name Resolution Rules

10 {*expected type (link name)* [partial]} The expected type for a *string*_expression in an interfacing pragma or in pragma Linker_Options is String.

10.a **Ramification:** There is no language-defined support for external or link names of type Wide_String, or of other string types. Implementations may, of course, have additional pragmas for that purpose. Note that allowing both String and Wide_String in the same pragma would cause ambiguities.

Legality Rules

11 {*convention*} The *convention*_identifier of an interfacing pragma shall be the name of a *convention*. The convention names are implementation defined, except for certain language-defined ones, such as Ada and Intrinsic, as explained in 6.3.1, "Conformance Rules". [Additional convention names generally represent the calling conventions of foreign languages, language implementations, or specific run-time models.] {*calling convention*} The convention of a callable entity is its *calling convention*.

11.a **Implementation defined:** Implementation-defined convention names.

11.b **Discussion:** We considered representing the convention names using an enumeration type declared in System. Then, *convention*_identifier would be changed to *convention*_name, and we would make its expected type be the enumeration type. We didn't do this because it seems to introduce extra complexity, and because the list of available languages is better represented as the list of children of package Interfaces — a more open-ended sort of list.

12 {*compatible (a type, with a convention)*} If *L* is a *convention*_identifier for a language, then a type T is said to be *compatible with convention L*, (alternatively, is said to be an *L-compatible type*) if any of the following conditions are met:

13 • T is declared in a language interface package corresponding to *L* and is defined to be *L*-compatible (see B.3, B.3.1, B.3.2, B.4, B.5),

14 • {*eligible (a type, for a convention)*} Convention *L* has been specified for T in a pragma Convention, and T is *eligible for convention L*; that is:

15 • T is an array type with either an unconstrained or statically-constrained first subtype, and its component type is *L*-compatible,

16 • T is a record type that has no discriminants and that only has components with statically-constrained subtypes, and each component type is *L*-compatible,

17 • T is an access-to-object type, and its designated type is *L*-compatible,

18 • T is an access-to-subprogram type, and its designated profile's parameter and result types are all *L*-compatible.

19 • T is derived from an *L*-compatible type,

20 • The implementation permits T as an *L*-compatible type.

20.a **Discussion:** For example, an implementation might permit Integer as a C-compatible type, though the C type to which it corresponds might be different in different environments.

21 If pragma Convention applies to a type, then the type shall either be compatible with or eligible for the convention specified in the pragma.

Ramification: If a type is derived from an *L*-compatible type, the derived type is by default *L*-compatible, but it is also permitted to specify pragma Convention for the derived type. 21.a

It is permitted to specify pragma Convention for an incomplete type, but in the complete declaration each component must be *L*-compatible. 21.b

If each component of a record type is *L*-compatible, then the record type itself is only *L*-compatible if it has a pragma Convention. 21.c

A pragma Import shall be the completion of a declaration. {*notwithstanding*} Notwithstanding any rule to the contrary, a pragma Import may serve as the completion of any kind of (explicit) declaration if supported by an implementation for that kind of declaration. If a completion is a pragma Import, then it shall appear in the same declarative_part, package_specification, task_definition or protected_definition as the declaration. For a library unit, it shall appear in the same compilation, before any subsequent compilation_units other than pragmas. If the local_name denotes more than one entity, then the pragma Import is the completion of all of them. 22

Discussion: For declarations of deferred constants and subprograms, we mention pragma Import explicitly as a possible completion. For other declarations that require completions, we ignore the possibility of pragma Import. Nevertheless, if an implementation chooses to allow a pragma Import to complete the declaration of a task, protected type, incomplete type, private type, etc., it may do so, and the normal completion is then not allowed for that declaration. 22.a

{*imported entity*} {*exported entity*} An entity specified as the Entity argument to a pragma Import (or pragma Export) is said to be *imported* (respectively, *exported*). 23

The declaration of an imported object shall not include an explicit initialization expression. [Default initializations are not performed.] 24

Proof: This follows from the “Notwithstanding ...” wording in the Dynamics Semantics paragraphs below. 24.a

The type of an imported or exported object shall be compatible with the convention specified in the corresponding pragma. 25

Ramification: This implies, for example, that importing an Integer object might be illegal, whereas importing an object of type Interfaces.C.int would be permitted. 25.a

For an imported or exported subprogram, the result and parameter types shall each be compatible with the convention specified in the corresponding pragma. 26

The external name and link name *string_*expressions of a pragma Import or Export, and the *string_*expression of a pragma Linker_Options, shall be static. 27

Static Semantics

{*representation pragma (Import)* [partial]} {*pragma, representation (Import)* [partial]} {*representation pragma (Export)* [partial]} {*pragma, representation (Export)* [partial]} {*representation pragma (Convention)* [partial]} {*pragma, representation (Convention)* [partial]} {*aspect of representation (convention, calling convention)* [partial]} {*convention (aspect of representation)*} Import, Export, and Convention pragmas are representation pragmas that specify the *convention* aspect of representation. {*aspect of representation (imported)* [partial]} {*imported (aspect of representation)*} {*aspect of representation (exported)* [partial]} {*exported (aspect of representation)*} In addition, Import and Export pragmas specify the *imported* and *exported* aspects of representation, respectively. 28

{*program unit pragma (Import)* [partial]} {*pragma, program unit (Import)* [partial]} {*program unit pragma (Export)* [partial]} {*pragma, program unit (Export)* [partial]} {*program unit pragma (Convention)* [partial]} {*pragma, program unit (Convention)* [partial]} An interfacing pragma is a program unit pragma when applied to a program unit (see 10.1.5). 29

30 An interfacing pragma defines the convention of the entity denoted by the local_name. The convention represents the calling convention or representation convention of the entity. For an access-to-subprogram type, it represents the calling convention of designated subprograms. In addition:

31 - A pragma Import specifies that the entity is defined externally (that is, outside the Ada program).

32 - A pragma Export specifies that the entity is used externally.

33 - A pragma Import or Export optionally specifies an entity's external name, link name, or both.

34 {*external name*} An *external name* is a string value for the name used by a foreign language program either for an entity that an Ada program imports, or for referring to an entity that an Ada program exports.

35 {*link name*} A *link name* is a string value for the name of an exported or imported entity, based on the conventions of the foreign language's compiler in interfacing with the system's linker tool.

36 The meaning of link names is implementation defined. If neither a link name nor the Address attribute of an imported or exported entity is specified, then a link name is chosen in an implementation-defined manner, based on the external name if one is specified.

36.a **Implementation defined:** The meaning of link names.

36.b **Ramification:** For example, an implementation might always prepend "_", and then pass it to the system linker.

36.c **Implementation defined:** The manner of choosing link names when neither the link name nor the address of an imported or exported entity is specified.

36.d **Ramification:** Normally, this will be the entity's defining name, or some simple transformation thereof.

37 Pragma Linker_Options has the effect of passing its string argument as a parameter to the system linker (if one exists), if the immediately enclosing compilation unit is included in the partition being linked. The interpretation of the string argument, and the way in which the string arguments from multiple Linker_Options pragmas are combined, is implementation defined.

37.a **Implementation defined:** The effect of pragma Linker_Options.

Dynamic Semantics

38 {*elaboration (declaration named by a pragma Import)* [partial]} {*notwithstanding*} Notwithstanding what this International Standard says elsewhere, the elaboration of a declaration denoted by the local_name of a pragma Import does not create the entity. Such an elaboration has no other effect than to allow the defining name to denote the external entity.

38.a **Ramification:** This implies that default initializations are skipped. (Explicit initializations are illegal.) For example, an imported access object is *not* initialized to **null**.

38.b Note that the local_name in a pragma Import might denote more than one declaration; in that case, the entity of all of those declarations will be the external entity.

38.c **Discussion:** This “notwithstanding” wording is better than saying “unless named by a pragma Import” on every definition of elaboration. It says we recognize the contradiction, and this rule takes precedence.

Erroneous Execution

38.1/2 {*AI95-00320-01*} {*erroneous execution (cause)* [partial]} It is the programmer's responsibility to ensure that the use of interfacing pragmas does not violate Ada semantics; otherwise, program execution is erroneous.

Implementation Advice

39 If an implementation supports pragma Export to a given language, then it should also allow the main subprogram to be written in that language. It should support some mechanism for invoking the elaboration of the Ada library units included in the system, and for invoking the finalization of the environment task. On typical systems, the recommended mechanism is to provide two subprograms

whose link names are "adainit" and "adafinal". Adainit should contain the elaboration code for library units. Adafinal should contain the finalization code. These subprograms should have no effect the second and subsequent time they are called. {*adainit*} {*adafinal*} {*Elaboration (of library units for a foreign language main subprogram)*} {*Finalization (of environment task for a foreign language main subprogram)*}

Implementation Advice: If pragma Export is supported for a language, the main program should be able to be written in that language. Subprograms named "adainit" and "adafinal" should be provided for elaboration and finalization of the environment task. 39.a.1/2

Ramification: For example, if the main subprogram is written in C, it can call adainit before the first call to an Ada subprogram, and adafinal after the last. 39.a

Automatic elaboration of preelaborated packages should be provided when pragma Export is supported. 40

Implementation Advice: Automatic elaboration of preelaborated packages should be provided when pragma Export is supported. 40.a.1/2

For each supported convention *L* other than Intrinsic, an implementation should support Import and Export pragmas for objects of *L*-compatible types and for subprograms, and pragma Convention for *L*-eligible types and for subprograms, presuming the other language has corresponding features. Pragma Convention need not be supported for scalar types. 41

Implementation Advice: For each supported convention *L* other than Intrinsic, pragmas Import and Export should be supported for objects of *L*-compatible types and for subprograms, and pragma Convention should be supported for *L*-eligible types and for subprograms. 41.a.1/2

Reason: Pragma Convention is not necessary for scalar types, since the language interface packages declare scalar types corresponding to those provided by the respective foreign languages. 41.a

Implementation Note: {*AI95-00114-01*} If an implementation supports interfacing to the C++ entities not supported by B.3, it should do so via the convention identifier C_Plus_Plus (in additional to any C++-implementation-specific ones). 41.b/2

Reason: {*AI95-00114-01*} The reason for giving the advice about C++ is to encourage uniformity among implementations, given that the name of the language is not syntactically legal as an identifier. 41.c/2

NOTES

1 Implementations may place restrictions on interfacing pragmas; for example, requiring each exported entity to be declared at the library level. 42

Proof: Arbitrary restrictions are allowed by 13.1. 42.a

Ramification: Such a restriction might be to disallow them altogether. Alternatively, the implementation might allow them only for certain kinds of entities, or only for certain conventions. 42.b

2 A pragma Import specifies the conventions for accessing external entities. It is possible that the actual entity is written in assembly language, but reflects the conventions of a particular language. For example, **pragma** Import(Ada, ...) can be used to interface to an assembly language routine that obeys the Ada compiler's calling conventions. 43

3 To obtain “call-back” to an Ada subprogram from a foreign language environment, **pragma** Convention should be specified both for the access-to-subprogram type and the specific subprogram(s) to which 'Access is applied. 44

4 It is illegal to specify more than one of Import, Export, or Convention for a given entity. 45

5 The local_name in an interfacing pragma can denote more than one entity in the case of overloading. Such a pragma applies to all of the denoted entities. 46

6 See also 13.8, “Machine Code Insertions”. 47

Ramification: The Intrinsic convention (see 6.3.1) implies that the entity is somehow “built in” to the implementation. Thus, it generally does not make sense for users to specify Intrinsic in a pragma Import. The intention is that only implementations will specify Intrinsic in a pragma Import. The language also defines certain subprograms to be Intrinsic. 47.a

Discussion: There are many imaginable interfacing pragmas that don't make any sense. For example, setting the Convention of a protected procedure to Ada is probably wrong. Rather than enumerating all such cases, however, we leave it up to implementations to decide what is sensible. 47.b

7 If both External_Name and Link_Name are specified for an Import or Export pragma, then the External_Name is ignored. 48

This paragraph was deleted.{*AI95-00320-01*} 49/2

Examples

50 *Example of interfacing pragmas:*

51
```
package Fortran_Library is
  function Sqrt (X : Float) return Float;
  function Exp  (X : Float) return Float;
private
  pragma Import(Fortran, Sqrt);
  pragma Import(Fortran, Exp);
end Fortran_Library;
```

Extensions to Ada 83

51.a {*extensions to Ada 83*} Interfacing pragmas are new to Ada 95. Pragma Import replaces Ada 83's pragma Interface. Existing implementations can continue to support pragma Interface for upward compatibility.

Wording Changes from Ada 95

51.b/2 {*8652/0058*} {*AI95-00036-01*} **Corrigendum:** Clarified that pragmas Import and Export work like a subprogram call; parameters cannot be omitted unless named notation is used. (Reordering is still not permitted, however.)

51.c/2 {*AI95-00320-01*} Added wording to say all bets are off if foreign code doesn't follow the semantics promised by the Ada specifications.

B.2 The Package Interfaces

1 Package Interfaces is the parent of several library packages that declare types and other entities useful for interfacing to foreign languages. It also contains some implementation-defined types that are useful across more than one language (in particular for interfacing to assembly language).

1.a **Implementation defined:** The contents of the visible part of package Interfaces and its language-defined descendants.

Static Semantics

2 The library package Interfaces has the following skeletal declaration:

3
```
package Interfaces is
   pragma Pure(Interfaces);
```
4
```
   type Integer_n is range -2**(n-1) .. 2**(n-1) - 1;  --2's complement
```
5
```
   type Unsigned_n is mod 2**n;
```
6
```
   function Shift_Left  (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Shift_Right (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Shift_Right_Arithmetic (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Rotate_Left  (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Rotate_Right (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   ...
end Interfaces;
```

Implementation Requirements

7 An implementation shall provide the following declarations in the visible part of package Interfaces:

8
- Signed and modular integer types of *n* bits, if supported by the target architecture, for each *n* that is at least the size of a storage element and that is a factor of the word size. The names of these types are of the form Integer_*n* for the signed types, and Unsigned_*n* for the modular types;

8.a **Ramification:** For example, for a typical 32-bit machine the corresponding types might be Integer_8, Unsigned_8, Integer_16, Unsigned_16, Integer_32, and Unsigned_32.

The wording above implies, for example, that Integer_16'Size = Unsigned_16'Size = 16. Unchecked conversions between same-Sized types will work as expected. 8.b

- {*shift*} {*rotate*} For each such modular type in Interfaces, shifting and rotating subprograms as specified in the declaration of Interfaces above. These subprograms are Intrinsic. They operate on a bit-by-bit basis, using the binary representation of the value of the operands to yield a binary representation for the result. The Amount parameter gives the number of bits by which to shift or rotate. For shifting, zero bits are shifted in, except in the case of Shift_Right_Arithmetic, where one bits are shifted in if Value is at least half the modulus. 9

Reason: We considered making shifting and rotating be primitive operations of all modular types. However, it is a design principle of Ada that all predefined operations should be operators (not functions named by identifiers). (Note that an early version of Ada had "**abs**" as an identifier, but it was changed to a reserved word operator before standardization of Ada 83.) This is important because the implicit declarations would hide non-overloadable declarations with the same name, whereas operators are always overloadable. Therefore, we would have had to make shift and rotate into reserved words, which would have been upward incompatible, or else invent new operator symbols, which seemed like too much mechanism. 9.a

- Floating point types corresponding to each floating point format fully supported by the hardware. 10

Implementation Note: The names for these floating point types are not specified. {*IEEE floating point arithmetic*} However, if IEEE arithmetic is supported, then the names should be IEEE_Float_32 and IEEE_Float_64 for single and double precision, respectively. 10.a

{*AI95-00204-01*} Support for interfacing to any foreign language is optional. However, an implementation shall not provide any attribute, library unit, or pragma having the same name as an attribute, library unit, or pragma (respectively) specified in the following clauses of this Annex unless the provided construct is either as specified in those clauses or is more limited in capability than that required by those clauses. A program that attempts to use an unsupported capability of this Annex shall either be identified by the implementation before run time or shall raise an exception at run time. 10.1/2

Discussion: The intent is that the same rules apply for language interfacing as apply for Specialized Needs Annexes. See 1.1.3 for a discussion of the purpose of these rules. 10.b/2

Implementation Permissions

An implementation may provide implementation-defined library units that are children of Interfaces, and may add declarations to the visible part of Interfaces in addition to the ones defined above. 11

Implementation defined: Implementation-defined children of package Interfaces. 11.a/2

{*AI95-00204-01*} A child package of package Interfaces with the name of a convention may be provided independently of whether the convention is supported by the pragma Convention and vice versa. Such a child package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in Interfaces. 11.1/2

Ramification: For example, package Interfaces.XYZ_Pascal might contain declarations of types that match the data types provided by the XYZ implementation of Pascal, so that it will be more convenient to pass parameters to a subprogram whose convention is XYZ_Pascal. 11.b/2

Implementation Advice

This paragraph was deleted.{*AI95-00204-01*} 12/2

This paragraph was deleted. 12.a/2

An implementation supporting an interface to C, COBOL, or Fortran should provide the corresponding package or packages described in the following clauses. 13

Implementation Advice: If an interface to C, COBOL, or Fortran is provided, the corresponding package or packages described in Annex B, “Interface to Other Languages” should also be provided. 13.a.1/2

13.a **Implementation Note:** The intention is that an implementation might support several implementations of the foreign language: Interfaces.This_Fortran and Interfaces.That_Fortran might both exist. The “default” implementation, overridable by the user, should be declared as a renaming:

13.b
```
    package Interfaces.Fortran renames Interfaces.This_Fortran;
```

Wording Changes from Ada 95

13.c/2 {*AI95-00204-01*} Clarified that interfacing to foreign languages is optional and has the same restrictions as a Specialized Needs Annex.

B.3 Interfacing with C and C++

1/2 {*8652/0059*} {*AI95-00131-01*} {*AI95-00376-01*} {*interface to C*} {*C interface*} The facilities relevant to interfacing with the C language and the corresponding subset of the C++ language are the package Interfaces.C and its children; support for the Import, Export, and Convention pragmas with *convention_*identifier C; and support for the Convention pragma with *convention_*identifier C_Pass_By_Copy.

2/2 {*AI95-00376-01*} The package Interfaces.C contains the basic types, constants and subprograms that allow an Ada program to pass scalars and strings to C and C++ functions. When this clause mentions a C entity, the reference also applies to the corresponding entity in C++.

Static Semantics

3 The library package Interfaces.C has the following declaration:

```
4     package Interfaces.C is
         pragma Pure(C);

5        -- Declarations based on C's <limits.h>

6        CHAR_BIT  : constant := implementation-defined;    -- typically 8
         SCHAR_MIN : constant := implementation-defined;    -- typically –128
         SCHAR_MAX : constant := implementation-defined;    -- typically 127
         UCHAR_MAX : constant := implementation-defined;    -- typically 255

7        -- Signed and Unsigned Integers
         type int   is range implementation-defined;
         type short is range implementation-defined;
         type long  is range implementation-defined;

8        type signed_char is range SCHAR_MIN .. SCHAR_MAX;
         for signed_char'Size use CHAR_BIT;

9        type unsigned       is mod implementation-defined;
         type unsigned_short is mod implementation-defined;
         type unsigned_long  is mod implementation-defined;

10       type unsigned_char is mod (UCHAR_MAX+1);
         for unsigned_char'Size use CHAR_BIT;

11       subtype plain_char is implementation-defined;

12       type ptrdiff_t is range implementation-defined;

13       type size_t is mod implementation-defined;

14       -- Floating Point

15       type C_float     is digits implementation-defined;

16       type double      is digits implementation-defined;

17       type long_double is digits implementation-defined;

18       -- Characters and Strings

19       type char is <implementation-defined character type>;

20/1  {8652/0060} {AI95-00037-01}    nul : constant char := implementation-defined;

21       function To_C   (Item : in Character) return char;
```

```
   function To_Ada (Item : in char) return Character;                          22

   type char_array is array (size_t range <>) of aliased char;                 23
   pragma Pack(char_array);
   for char_array'Component_Size use CHAR_BIT;

   function Is_Nul_Terminated (Item : in char_array) return Boolean;           24

   function To_C   (Item       : in String;                                    25
                    Append_Nul : in Boolean := True)
      return char_array;

   function To_Ada (Item     : in char_array;                                  26
                    Trim_Nul : in Boolean := True)
      return String;

   procedure To_C (Item       : in  String;                                    27
                   Target     : out char_array;
                   Count      : out size_t;
                   Append_Nul : in  Boolean := True);

   procedure To_Ada (Item     : in  char_array;                                28
                     Target   : out String;
                     Count    : out Natural;
                     Trim_Nul : in  Boolean := True);

   -- Wide Character and Wide String                                           29

{8652/0060} {AI95-00037-01}    type wchar_t is <implementation-defined character type>;   30/1

{8652/0060} {AI95-00037-01}    wide_nul : constant wchar_t := implementation-defined;     31/1

   function To_C   (Item : in Wide_Character) return wchar_t;                  32
   function To_Ada (Item : in wchar_t       ) return Wide_Character;

   type wchar_array is array (size_t range <>) of aliased wchar_t;             33

   pragma Pack(wchar_array);                                                   34

   function Is_Nul_Terminated (Item : in wchar_array) return Boolean;          35

   function To_C   (Item       : in Wide_String;                               36
                    Append_Nul : in Boolean := True)
      return wchar_array;

   function To_Ada (Item     : in wchar_array;                                 37
                    Trim_Nul : in Boolean := True)
      return Wide_String;

   procedure To_C (Item       : in  Wide_String;                               38
                   Target     : out wchar_array;
                   Count      : out size_t;
                   Append_Nul : in  Boolean := True);

   procedure To_Ada (Item     : in  wchar_array;                               39
                     Target   : out Wide_String;
                     Count    : out Natural;
                     Trim_Nul : in  Boolean := True);

{AI95-00285-01}     -- ISO/IEC 10646:2003 compatible types defined by ISO/IEC TR 19769:2004.   39.1/2

{AI95-00285-01}    type char16_t is <implementation-defined character type>;   39.2/2

   char16_nul : constant char16_t := implementation-defined;                   39.3/2

   function To_C (Item : in Wide_Character) return char16_t;                   39.4/2
   function To_Ada (Item : in char16_t) return Wide_Character;

   type char16_array is array (size_t range <>) of aliased char16_t;           39.5/2

   pragma Pack(char16_array);                                                  39.6/2

   function Is_Nul_Terminated (Item : in char16_array) return Boolean;         39.7/2
   function To_C (Item       : in Wide_String;
                  Append_Nul : in Boolean := True)
      return char16_array;

   function To_Ada (Item     : in char16_array;                                39.8/2
                    Trim_Nul : in Boolean := True)
      return Wide_String;
```

39.9/2
```
   procedure To_C (Item       : in  Wide_String;
                   Target     : out char16_array;
                   Count      : out size_t;
                   Append_Nul : in  Boolean := True);
```

39.10/2
```
   procedure To_Ada (Item     : in  char16_array;
                     Target   : out Wide_String;
                     Count    : out Natural;
                     Trim_Nul : in  Boolean := True);
```

39.11/2
```
{AI95-00285-01}    type char32_t is <implementation-defined character type>;
```

39.12/2
```
   char32_nul : constant char32_t := implementation-defined;
```

39.13/2
```
   function To_C (Item : in Wide_Wide_Character) return char32_t;
   function To_Ada (Item : in char32_t) return Wide_Wide_Character;
```

39.14/2
```
   type char32_array is array (size_t range <>) of aliased char32_t;
```

39.15/2
```
   pragma Pack(char32_array);
```

39.16/2
```
   function Is_Nul_Terminated (Item : in char32_array) return Boolean;
   function To_C (Item       : in Wide_Wide_String;
                  Append_Nul : in Boolean := True)
      return char32_array;
```

39.17/2
```
   function To_Ada (Item     : in char32_array;
                    Trim_Nul : in Boolean := True)
      return Wide_Wide_String;
```

39.18/2
```
   procedure To_C (Item       : in  Wide_Wide_String;
                   Target     : out char32_array;
                   Count      : out size_t;
                   Append_Nul : in  Boolean := True);
```

39.19/2
```
   procedure To_Ada (Item     : in  char32_array;
                     Target   : out Wide_Wide_String;
                     Count    : out Natural;
                     Trim_Nul : in  Boolean := True);
```

40
```
   Terminator_Error : exception;
```

41
```
end Interfaces.C;
```

41.a.1/2 **Implementation defined:** The definitions of certain types and constants in Interfaces.C.

42 Each of the types declared in Interfaces.C is C-compatible.

43/2 {*AI95-00285-01*} The types int, short, long, unsigned, ptrdiff_t, size_t, double, char, wchar_t, char16_t, and char32_t correspond respectively to the C types having the same names. The types signed_char, unsigned_short, unsigned_long, unsigned_char, C_float, and long_double correspond respectively to the C types signed char, unsigned short, unsigned long, unsigned char, float, and long double.

43.a/2 **Discussion:** The C types wchar_t and char16_t seem to be the same. However, wchar_t has an implementation-defined size, whereas char16_t is guaranteed to be an unsigned type of at least 16 bits. Also, char16_t and char32_t are encouraged to have UTF-16 and UTF-32 representations; that means that they are not directly the same as the Ada types, which most likely don't use any UTF encoding.

44 The type of the subtype plain_char is either signed_char or unsigned_char, depending on the C implementation.

45
```
function To_C   (Item : in Character) return char;
function To_Ada (Item : in char     ) return Character;
```

46 The functions To_C and To_Ada map between the Ada type Character and the C type char.

46.a.1/1 **Implementation Note:** {*8652/0114*} {*AI95-00038-01*} The To_C and To_Ada functions map between corresponding characters, not necessarily between characters with the same internal representation. Corresponding characters are characters defined by the same enumeration literal, if such exist; otherwise, the correspondence is unspecified.{*Unspecified* [partial]}

46.a.2/1 The following definition is equivalent to the above summary:

46.a.3/1
```
To_C (Latin_1_Char) = char'Value(Character'Image(Latin_1_Char))
```

provided that char'Value does not raise an exception; otherwise the result is unspecified.

```
To_Ada (Native_C_Char) = Character'Value(char'Image(Native_C_Char))
```
46.a.4/1

provided that Character'Value does not raise an exception; otherwise the result is unspecified.

```
function Is_Nul_Terminated (Item : in char_array) return Boolean;
```
47

48 The result of Is_Nul_Terminated is True if Item contains nul, and is False otherwise.

```
function To_C   (Item : in String;     Append_Nul : in Boolean := True)
   return char_array;

function To_Ada (Item : in char_array; Trim_Nul   : in Boolean := True)
   return String;
```
49

50/2 {*AI95-00258-01*} The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True. If Append_Nul is False and Item'Length is 0, then To_C propagates Constraint_Error.

51 The result of To_Ada is a String whose length is Item'Length (if Trim_Nul is False) or the length of the slice of Item preceding the first nul (if Trim_Nul is True). The lower bound of the result is 1. If Trim_Nul is False, then for each component Item(I) the corresponding component in the result is To_Ada applied to Item(I). If Trim_Nul is True, then for each component Item(I) before the first nul the corresponding component in the result is To_Ada applied to Item(I). The function propagates Terminator_Error if Trim_Nul is True and Item does not contain nul.

```
procedure To_C (Item       : in  String;
                Target     : out char_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char_array;
                  Target   : out String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```
52

53 For procedure To_C, each element of Item is converted (via the To_C function) to a char, which is assigned to the corresponding element of Target. If Append_Nul is True, nul is then assigned to the next element of Target. In either case, Count is set to the number of Target elements assigned. {*Constraint_Error (raised by failure of run-time check)*} If Target is not long enough, Constraint_Error is propagated.

54 For procedure To_Ada, each element of Item (if Trim_Nul is False) or each element of Item preceding the first nul (if Trim_Nul is True) is converted (via the To_Ada function) to a Character, which is assigned to the corresponding element of Target. Count is set to the number of Target elements assigned. {*Constraint_Error (raised by failure of run-time check)*} If Target is not long enough, Constraint_Error is propagated. If Trim_Nul is True and Item does not contain nul, then Terminator_Error is propagated.

```
function Is_Nul_Terminated (Item : in wchar_array) return Boolean;
```
55

56 The result of Is_Nul_Terminated is True if Item contains wide_nul, and is False otherwise.

```
function To_C   (Item : in Wide_Character) return wchar_t;
function To_Ada (Item : in wchar_t       ) return Wide_Character;
```
57

58 To_C and To_Ada provide the mappings between the Ada and C wide character types.

59
```
function To_C   (Item       : in Wide_String;
                 Append_Nul : in Boolean := True)
   return wchar_array;

function To_Ada (Item     : in wchar_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out wchar_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  wchar_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

60 The To_C and To_Ada subprograms that convert between Wide_String and wchar_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that wide_nul is used instead of nul.

60.1/2
```
function Is_Nul_Terminated (Item : in char16_array) return Boolean;
```

60.2/2 {*AI95-00285-01*} The result of Is_Nul_Terminated is True if Item contains char16_nul, and is False otherwise.

60.3/2
```
function To_C (Item : in Wide_Character) return char16_t;
function To_Ada (Item : in char16_t ) return Wide_Character;
```

60.4/2 {*AI95-00285-01*} To_C and To_Ada provide mappings between the Ada and C 16-bit character types.

60.5/2
```
function To_C (Item       : in Wide_String;
               Append_Nul : in Boolean := True)
   return char16_array;

function To_Ada (Item     : in char16_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out char16_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char16_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

60.6/2 {*AI95-00285-01*} The To_C and To_Ada subprograms that convert between Wide_String and char16_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char16_nul is used instead of nul.

60.7/2
```
function Is_Nul_Terminated (Item : in char32_array) return Boolean;
```

60.8/2 {*AI95-00285-01*} The result of Is_Nul_Terminated is True if Item contains char16_nul, and is False otherwise.

60.9/2
```
function To_C (Item : in Wide_Wide_Character) return char32_t;
function To_Ada (Item : in char32_t ) return Wide_Wide_Character;
```

60.10/2 {*AI95-00285-01*} To_C and To_Ada provide mappings between the Ada and C 32-bit character types.

60.11/2
```
function To_C (Item       : in Wide_Wide_String;
               Append_Nul : in Boolean := True)
   return char32_array;

function To_Ada (Item     : in char32_array;
                 Trim_Nul : in Boolean := True)
   return Wide_Wide_String;

procedure To_C (Item       : in  Wide_Wide_String;
                Target     : out char32_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char32_array;
                  Target   : out Wide_Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

60.12/2 {*AI95-00285-01*} The To_C and To_Ada subprograms that convert between Wide_Wide_String and char32_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char32_nul is used instead of nul.

60.a **Discussion:** The Interfaces.C package provides an implementation-defined character type, char, designed to model the C run-time character set, and mappings between the types char and Character.

60.b One application of the C interface package is to compose a C string and pass it to a C function. One way to do this is for the programmer to declare an object that will hold the C array, and then pass this array to the C function. This is realized via the type char_array:

60.c
```
type char_array is array (size_t range <>) of Char;
```

60.d The programmer can declare an Ada String, convert it to a char_array, and pass the char_array as actual parameter to the C function that is expecting a char *.

60.e An alternative approach is for the programmer to obtain a C char pointer from an Ada String (or from a char_array) by invoking an allocation function. The package Interfaces.C.Strings (see below) supplies the needed facilities, including a private type chars_ptr that corresponds to C's char *, and two allocation functions. To avoid storage leakage, a Free procedure releases the storage that was allocated by one of these allocate functions.

60.f It is typical for a C function that deals with strings to adopt the convention that the string is delimited by a nul char. The C interface packages support this convention. A constant nul of type Char is declared, and the function Value(Chars_Ptr) in Interfaces.C.Strings returns a char_array up to and including the first nul in the array that the chars_ptr points to. The Allocate_Chars function allocates an array that is nul terminated.

60.g Some C functions that deal with strings take an explicit length as a parameter, thus allowing strings to be passed that contain nul as a data element. Other C functions take an explicit length that is an upper bound: the prefix of the string up to the char before nul, or the prefix of the given length, is used by the function, whichever is shorter. The C Interface packages support calling such functions.

60.13/1 {*8652/0059*} {*AI95-00131-01*} A Convention pragma with *convention_*identifier C_Pass_By_Copy shall only be applied to a type.

60.14/2 {*8652/0059*} {*AI95-00131-01*} {*AI95-00216-01*} The eligibility rules in B.1 do not apply to convention C_Pass_By_Copy. Instead, a type T is eligible for convention C_Pass_By_Copy if T is an unchecked union type or if T is a record type that has no discriminants and that only has components with statically constrained subtypes, and each component is C-compatible.

60.15/1 {*8652/0059*} {*AI95-00131-01*} If a type is C_Pass_By_Copy-compatible then it is also C-compatible.

Implementation Requirements

61/1 {*8652/0059*} {*AI95-00131-01*} An implementation shall support pragma Convention with a C *convention_*identifier for a C-eligible type (see B.1). An implementation shall support pragma Convention with a C_Pass_By_Copy *convention_*identifier for a C_Pass_By_Copy-eligible type.

Implementation Permissions

62 An implementation may provide additional declarations in the C interface packages.

Implementation Advice

62.1/2 {*8652/0060*} {*AI95-00037-01*} {*AI95-00285-01*} The constants nul, wide_nul, char16_nul, and char32_nul should have a representation of zero.

62.a/2 **Implementation Advice:** The constants nul, wide_nul, char16_nul, and char32_nul in package Interfaces.C should have a representation of zero.

63 An implementation should support the following interface correspondences between Ada and C.

64 • An Ada procedure corresponds to a void-returning C function.

64.a **Discussion:** The programmer can also choose an Ada procedure when the C function returns an int that is to be discarded.

65 • An Ada function corresponds to a non-void C function.

66 • An Ada **in** scalar parameter is passed as a scalar argument to a C function.

67 • An Ada **in** parameter of an access-to-object type with designated type T is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.

68 • An Ada **access** T parameter, or an Ada **out** or **in out** parameter of an elementary type T, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T. In the case of an elementary **out** or **in out** parameter, a pointer to a temporary copy is used to preserve by-copy semantics.

68.1/2 • {*8652/0059*} {*AI95-00131-01*} {*AI95-00343-01*} An Ada parameter of a (record) type T of convention C_Pass_By_Copy, of mode **in**, is passed as a t argument to a C function, where t is the C struct corresponding to the Ada type T.

69/2 • {*8652/0059*} {*AI95-00131-01*} {*AI95-00343-01*} An Ada parameter of a record type T, of any mode, other than an **in** parameter of a type of convention C_Pass_By_Copy, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.

70 • An Ada parameter of an array type with component type T, of any mode, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.

71 • An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification.

71.1/2 {*AI95-00337-01*} An Ada parameter of a private type is passed as specified for the full view of the type.

71.a/2 **Implementation Advice:** If C interfacing is supported, the interface correspondences between Ada and C should be supported.

NOTES

72 8 Values of type char_array are not implicitly terminated with nul. If a char_array is to be passed as a parameter to an imported C function requiring nul termination, it is the programmer's responsibility to obtain this effect.

73 9 To obtain the effect of C's sizeof(item_type), where Item_Type is the corresponding Ada type, evaluate the expression: size_t(Item_Type'Size/CHAR_BIT).

74/2 *This paragraph was deleted.*{*AI95-00216-01*}

75 10 A C function that takes a variable number of arguments can correspond to several Ada subprograms, taking various specific numbers and types of parameters.

Examples

Example of using the Interfaces.C package: 76

```
--Calling the C Library Function strcpy                                                    77
with Interfaces.C;
procedure Test is
   package C renames Interfaces.C;
   use type C.char_array;
   -- Call <string.h>strcpy:
   -- C definition of strcpy:  char *strcpy(char *s1, const char *s2);
   --    This function copies the string pointed to by s2 (including the terminating null character)
   --     into the array pointed to by s1. If copying takes place between objects that overlap,
   --     the behavior is undefined. The strcpy function returns the value of s1.

   -- Note: since the C function's return value is of no interest, the Ada interface is a procedure   78
   procedure Strcpy (Target : out C.char_array;
                     Source : in  C.char_array);

   pragma Import(C, Strcpy, "strcpy");                                                     79

   Chars1 :  C.char_array(1..20);                                                          80
   Chars2 :  C.char_array(1..20);

begin                                                                                      81
   Chars2(1..6) := "qwert" & C.nul;

   Strcpy(Chars1, Chars2);                                                                 82

-- Now Chars1(1..6) = "qwert" & C.Nul                                                      83

end Test;                                                                                  84
```

Incompatibilities With Ada 95

{*AI95-00285-01*} {*incompatibilities with Ada 95*} Types char16_t and char32_t and their related types and operations are newly added to Interfaces.C. If Interfaces.C is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Interfaces.C is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 84.a/2

Extensions to Ada 95

{*8652/0059*} {*AI95-00131-01*} {*extensions to Ada 95*} **Corrigendum:** Convention C_Pass_By_Copy is new. 84.b/2

Wording Changes from Ada 95

{*8652/0060*} {*AI95-00037-01*} **Corrigendum:** Clarified the intent for Nul and Wide_Nul. 84.c/2

{*AI95-00216-01*} Specified that an unchecked union type (see B.3.3) is eligible for convention C_Pass_By_Copy. 84.d/2

{*AI95-00258-01*} Specified what happens if the To_C function tries to return a null string. 84.e/2

{*AI95-00337-01*} Clarified that the interface correspondences also apply to private types whose full types have the specified characteristics. 84.f/2

{*AI95-00343-01*} Clarified that a type must have convention C_Pass_By_Copy in order to be passed by copy (not just a type that could have that convention). 84.g/2

{*AI95-00376-01*} Added wording to make it clear that these facilities can also be used with C++. 84.h/2

B.3.1 The Package Interfaces.C.Strings

The package Interfaces.C.Strings declares types and subprograms allowing an Ada program to allocate, reference, update, and free C-style strings. In particular, the private type chars_ptr corresponds to a common use of “char *” in C programs, and an object of this type can be passed to a subprogram to which pragma Import(C,...) has been applied, and for which “char *” is the type of the argument of the C function. 1

Static Semantics

The library package Interfaces.C.Strings has the following declaration: 2

3
```
package Interfaces.C.Strings is
   pragma Preelaborate(Strings);
```
4
```
   type char_array_access is access all char_array;
```
5/2
```
{AI95-00161-01}    type chars_ptr is private;
   pragma Preelaborable_Initialization(chars_ptr);
```
6/2
```
{AI95-00276-01}    type chars_ptr_array is array (size_t range <>) of aliased
chars_ptr;
```
7
```
   Null_Ptr : constant chars_ptr;
```
8
```
   function To_Chars_Ptr (Item      : in char_array_access;
                          Nul_Check : in Boolean := False)
      return chars_ptr;
```
9
```
   function New_Char_Array (Chars   : in char_array) return chars_ptr;
```
10
```
   function New_String (Str : in String) return chars_ptr;
```
11
```
   procedure Free (Item : in out chars_ptr);
```
12
```
   Dereference_Error : exception;
```
13
```
   function Value (Item : in chars_ptr) return char_array;
```
14
```
   function Value (Item : in chars_ptr; Length : in size_t)
      return char_array;
```
15
```
   function Value (Item : in chars_ptr) return String;
```
16
```
   function Value (Item : in chars_ptr; Length : in size_t)
      return String;
```
17
```
   function Strlen (Item : in chars_ptr) return size_t;
```
18
```
   procedure Update (Item   : in chars_ptr;
                     Offset : in size_t;
                     Chars  : in char_array;
                     Check  : in Boolean := True);
```
19
```
   procedure Update (Item   : in chars_ptr;
                     Offset : in size_t;
                     Str    : in String;
                     Check  : in Boolean := True);
```
20
```
   Update_Error : exception;
```
21
```
private
   ... -- not specified by the language
end Interfaces.C.Strings;
```

21.a **Discussion:** The string manipulation types and subprograms appear in a child of Interfaces.C versus being there directly, since it is useful to have Interfaces.C specified as pragma Pure.

21.b Differently named functions New_String and New_Char_Array are declared, since if there were a single overloaded function a call with a string literal as actual parameter would be ambiguous.

22 The type chars_ptr is C-compatible and corresponds to the use of C's “char *” for a pointer to the first char in a char array terminated by nul. When an object of type chars_ptr is declared, its value is by default set to Null_Ptr, unless the object is imported (see B.1).

22.a **Discussion:** The type char_array_access is not necessarily C-compatible, since an object of this type may carry “dope” information. The programmer should convert from char_array_access to chars_ptr for objects imported from, exported to, or passed to C.

23
```
function To_Chars_Ptr (Item      : in char_array_access;
                       Nul_Check : in Boolean := False)
   return chars_ptr;
```

24/1 {*8652/0061*} {*AI95-00140-01*} If Item is **null**, then To_Chars_Ptr returns Null_Ptr. If Item is not **null**, Nul_Check is True, and Item.**all** does not contain nul, then the function propagates Terminator_Error; otherwise To_Chars_Ptr performs a pointer conversion with no allocation of memory.

```
function New_Char_Array (Chars   : in char_array) return chars_ptr;
```
25

26 This function returns a pointer to an allocated object initialized to Chars(Chars'First .. Index) & nul, where

- 27 Index = Chars'Last if Chars does not contain nul, or
- 28 Index is the smallest size_t value I such that Chars(I+1) = nul.

28.1 Storage_Error is propagated if the allocation fails.

```
function New_String (Str : in String) return chars_ptr;
```
29

30 This function is equivalent to New_Char_Array(To_C(Str)).

```
procedure Free (Item : in out chars_ptr);
```
31

32 If Item is Null_Ptr, then Free has no effect. Otherwise, Free releases the storage occupied by Value(Item), and resets Item to Null_Ptr.

```
function Value (Item : in chars_ptr) return char_array;
```
33

34 If Item = Null_Ptr then Value propagates Dereference_Error. Otherwise Value returns the prefix of the array of chars pointed to by Item, up to and including the first nul. The lower bound of the result is 0. If Item does not point to a nul-terminated string, then execution of Value is erroneous.

```
function Value (Item : in chars_ptr; Length : in size_t)
   return char_array;
```
35

36/1 {*8652/0062*} {*AI95-00139-01*} If Item = Null_Ptr then Value propagates Dereference_Error. Otherwise Value returns the shorter of two arrays, either the first Length chars pointed to by Item, or Value(Item). The lower bound of the result is 0. If Length is 0, then Value propagates Constraint_Error.

36.a **Ramification:** Value(New_Char_Array(Chars)) = Chars if Chars does not contain nul; else Value(New_Char_Array(Chars)) is the prefix of Chars up to and including the first nul.

```
function Value (Item : in chars_ptr) return String;
```
37

38 Equivalent to To_Ada(Value(Item), Trim_Nul=>True).

```
function Value (Item : in chars_ptr; Length : in size_t)
   return String;
```
39

40/1 {*8652/0063*} {*AI95-00177-01*} Equivalent to To_Ada(Value(Item, Length) & nul, Trim_Nul=>True).

```
function Strlen (Item : in chars_ptr) return size_t;
```
41

42 Returns *Val*'Length–1 where *Val* = Value(Item); propagates Dereference_Error if Item = Null_Ptr.

42.a **Ramification:** Strlen returns the number of chars in the array pointed to by Item, up to and including the char immediately before the first nul.

42.b Strlen has the same possibility for erroneous execution as Value, in cases where the string has not been nul-terminated.

42.c Strlen has the effect of C's strlen function.

```
procedure Update (Item   : in chars_ptr;
                  Offset : in size_t;
                  Chars  : in char_array;
                  Check  : Boolean := True);
```
43

44/1 {*8652/0064*} {*AI95-00039-01*} If Item = Null_Ptr, then Update propagates Dereference_Error. Otherwise, this procedure updates the value pointed to by Item, starting at position Offset, using

Chars as the data to be copied into the array. Overwriting the nul terminator, and skipping with the Offset past the nul terminator, are both prevented if Check is True, as follows:

45
- Let N = Strlen(Item). If Check is True, then:

46
  - If Offset+Chars'Length>N, propagate Update_Error.

47
  - Otherwise, overwrite the data in the array pointed to by Item, starting at the char at position Offset, with the data in Chars.

48
- If Check is False, then processing is as above, but with no check that Offset+Chars'Length>N.

48.a **Ramification:** If Chars contains nul, Update's effect may be to “shorten” the pointed-to char array.

49
```
procedure Update (Item   : in chars_ptr;
                  Offset : in size_t;
                  Str    : in String;
                  Check  : in Boolean := True);
```

50/2 {*AI95-00242-01*} Equivalent to Update(Item, Offset, To_C(Str, Append_Nul => False), Check).

50.a/2 **Discussion:** {*AI95-00242-01*} To truncate the Item to the length of Str, use Update(Item, Offset, To_C(Str), Check) instead of Update(Item, Offset, Str, Check). Note that when truncating Item, Item must be longer than Str.

Erroneous Execution

51 {*erroneous execution (cause)* [partial]} Execution of any of the following is erroneous if the Item parameter is not null_ptr and Item does not point to a nul-terminated array of chars.

52
- a Value function not taking a Length parameter,

53
- the Free procedure,

54
- the Strlen function.

55 {*erroneous execution (cause)* [partial]} Execution of Free(X) is also erroneous if the chars_ptr X was not returned by New_Char_Array or New_String.

56 {*erroneous execution (cause)* [partial]} Reading or updating a freed char_array is erroneous.

57 {*erroneous execution (cause)* [partial]} Execution of Update is erroneous if Check is False and a call with Check equal to True would have propagated Update_Error.

NOTES

58 11 New_Char_Array and New_String might be implemented either through the allocation function from the C environment (“malloc”) or through Ada dynamic memory allocation (“new”). The key points are

59
- the returned value (a chars_ptr) is represented as a C “char *” so that it may be passed to C functions;

60
- the allocated object should be freed by the programmer via a call of Free, not by a called C function.

Inconsistencies With Ada 95

60.a/2 {*AI95-00242-01*} {*inconsistencies with Ada 95*} **Amendment Correction:** Update for a String parameter is now defined to not add a nul character. It did add a nul in Ada 95. This means that programs that used this behavior of Update to truncate a string will no longer work (the string will not be truncated). This change makes Update for a string consistent with Update for a char_array (no implicit nul is added to the end of a char_array).

Extensions to Ada 95

60.b/2 {*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Added pragma Preelaborable_Initialization to type chars_ptr, so that it can be used in preelaborated units.

60.c/2 {*AI95-00276-01*} **Amendment Correction:** The components of chars_ptr_array are aliased so that it can be used to instantiate Interfaces.C.Pointers (that is its intended purpose, which is otherwise mysterious as it has no operations).

Wording Changes from Ada 95

60.d/2 {*8652/0061*} {*AI95-00140-01*} **Corrigendum:** Fixed the missing semantics of To_Char_Ptr when Nul_Check is False.

{*8652/0062*} {*AI95-00139-01*} **Corrigendum:** Fixed the missing semantics of Value when the Length is 0. 60.e/2

{*8652/0063*} {*AI95-00177-01*} **Corrigendum:** Corrected the definition of Value to avoid raising Terminator_Error. 60.f/2

{*8652/0064*} {*AI95-00039-01*} **Corrigendum:** Fixed the missing semantics of Update when Item is Null_Ptr. 60.g/2

B.3.2 The Generic Package Interfaces.C.Pointers

The generic package Interfaces.C.Pointers allows the Ada programmer to perform C-style operations on pointers. It includes an access type Pointer, Value functions that dereference a Pointer and deliver the designated array, several pointer arithmetic operations, and “copy” procedures that copy the contents of a source pointer into the array designated by a destination pointer. As in C, it treats an object Ptr of type Pointer as a pointer to the first element of an array, so that for example, adding 1 to Ptr yields a pointer to the second element of the array. 1

The generic allows two styles of usage: one in which the array is terminated by a special terminator element; and another in which the programmer needs to keep track of the length. 2

Static Semantics

The generic library package Interfaces.C.Pointers has the following declaration: 3

4
```
generic
   type Index is (<>);
   type Element is private;
   type Element_Array is array (Index range <>) of aliased Element;
   Default_Terminator : Element;
package Interfaces.C.Pointers is
   pragma Preelaborate(Pointers);
```

5
```
   type Pointer is access all Element;
```

6
```
   function Value(Ref        : in Pointer;
                  Terminator : in Element := Default_Terminator)
      return Element_Array;
```

7
```
   function Value(Ref    : in Pointer;
                  Length : in ptrdiff_t)
      return Element_Array;
```

8
```
   Pointer_Error : exception;
```

9
```
   -- C-style Pointer arithmetic
```

10
```
   function "+" (Left : in Pointer;   Right : in ptrdiff_t) return Pointer;
   function "+" (Left : in ptrdiff_t; Right : in Pointer)   return Pointer;
   function "-" (Left : in Pointer;   Right : in ptrdiff_t) return Pointer;
   function "-" (Left : in Pointer;   Right : in Pointer) return ptrdiff_t;
```

11
```
   procedure Increment (Ref : in out Pointer);
   procedure Decrement (Ref : in out Pointer);
```

12
```
   pragma Convention (Intrinsic, "+");
   pragma Convention (Intrinsic, "-");
   pragma Convention (Intrinsic, Increment);
   pragma Convention (Intrinsic, Decrement);
```

13
```
   function Virtual_Length (Ref        : in Pointer;
                            Terminator : in Element := Default_Terminator)
      return ptrdiff_t;
```

14
```
   procedure Copy_Terminated_Array
      (Source     : in Pointer;
       Target     : in Pointer;
       Limit      : in ptrdiff_t := ptrdiff_t'Last;
       Terminator : in Element :=  Default_Terminator);
```

15
```
   procedure Copy_Array (Source  : in Pointer;
                         Target  : in Pointer;
                         Length  : in ptrdiff_t);
```

16
```
   end Interfaces.C.Pointers;
```

17 The type Pointer is C-compatible and corresponds to one use of C's “Element *”. An object of type Pointer is interpreted as a pointer to the initial Element in an Element_Array. Two styles are supported:

18 • Explicit termination of an array value with Default_Terminator (a special terminator value);

19 • Programmer-managed length, with Default_Terminator treated simply as a data element.

20
```
   function Value(Ref        : in Pointer;
                  Terminator : in Element := Default_Terminator)
      return Element_Array;
```

21 This function returns an Element_Array whose value is the array pointed to by Ref, up to and including the first Terminator; the lower bound of the array is Index'First. Interfaces.C.Strings.Dereference_Error is propagated if Ref is **null**.

22
```
   function Value(Ref    : in Pointer;
                  Length : in ptrdiff_t)
      return Element_Array;
```

23 This function returns an Element_Array comprising the first Length elements pointed to by Ref. The exception Interfaces.C.Strings.Dereference_Error is propagated if Ref is **null**.

24 The "+" and "–" functions perform arithmetic on Pointer values, based on the Size of the array elements. In each of these functions, Pointer_Error is propagated if a Pointer parameter is **null**.

25
```
   procedure Increment (Ref : in out Pointer);
```

26 Equivalent to Ref := Ref+1.

27
```
   procedure Decrement (Ref : in out Pointer);
```

28 Equivalent to Ref := Ref–1.

29
```
   function Virtual_Length (Ref        : in Pointer;
                            Terminator : in Element := Default_Terminator)
      return ptrdiff_t;
```

30 Returns the number of Elements, up to the one just before the first Terminator, in Value(Ref, Terminator).

31
```
   procedure Copy_Terminated_Array
      (Source     : in Pointer;
       Target     : in Pointer;
       Limit      : in ptrdiff_t := ptrdiff_t'Last;
       Terminator : in Element := Default_Terminator);
```

32 This procedure copies Value(Source, Terminator) into the array pointed to by Target; it stops either after Terminator has been copied, or the number of elements copied is Limit, whichever occurs first. Dereference_Error is propagated if either Source or Target is **null**.

32.a **Ramification:** It is the programmer's responsibility to ensure that elements are not copied beyond the logical length of the target array.

32.b **Implementation Note:** The implementation has to take care to check the Limit first.

33
```
   procedure Copy_Array (Source  : in Pointer;
                         Target  : in Pointer;
                         Length  : in ptrdiff_t);
```

34 This procedure copies the first Length elements from the array pointed to by Source, into the array pointed to by Target. Dereference_Error is propagated if either Source or Target is **null**.

Erroneous Execution

{*erroneous execution (cause)* [partial]} It is erroneous to dereference a Pointer that does not designate an aliased Element. 35

Discussion: Such a Pointer could arise via "+", "–", Increment, or Decrement. 35.a

{*erroneous execution (cause)* [partial]} Execution of Value(Ref, Terminator) is erroneous if Ref does not designate an aliased Element in an Element_Array terminated by Terminator. 36

{*erroneous execution (cause)* [partial]} Execution of Value(Ref, Length) is erroneous if Ref does not designate an aliased Element in an Element_Array containing at least Length Elements between the designated Element and the end of the array, inclusive. 37

{*erroneous execution (cause)* [partial]} Execution of Virtual_Length(Ref, Terminator) is erroneous if Ref does not designate an aliased Element in an Element_Array terminated by Terminator. 38

{*erroneous execution (cause)* [partial]} Execution of Copy_Terminated_Array(Source, Target, Limit, Terminator) is erroneous in either of the following situations: 39

- Execution of both Value(Source, Terminator) and Value(Source, Limit) are erroneous, or 40
- Copying writes past the end of the array containing the Element designated by Target. 41

{*erroneous execution (cause)* [partial]} Execution of Copy_Array(Source, Target, Length) is erroneous if either Value(Source, Length) is erroneous, or copying writes past the end of the array containing the Element designated by Target. 42

NOTES

12 To compose a Pointer from an Element_Array, use 'Access on the first element. For example (assuming appropriate instantiations): 43

44
```
Some_Array   : Element_Array(0..5) ;
Some_Pointer : Pointer := Some_Array(0)'Access;
```

Examples

Example of Interfaces.C.Pointers: 45

46
```
with Interfaces.C.Pointers;
with Interfaces.C.Strings;
procedure Test_Pointers is
   package C renames Interfaces.C;
   package Char_Ptrs is
      new C.Pointers (Index              => C.size_t,
                      Element            => C.char,
                      Element_Array      => C.char_array,
                      Default_Terminator => C.nul);
```

47
```
   use type Char_Ptrs.Pointer;
   subtype Char_Star is Char_Ptrs.Pointer;
```

48
```
   procedure Strcpy (Target_Ptr, Source_Ptr : Char_Star) is
      Target_Temp_Ptr : Char_Star := Target_Ptr;
      Source_Temp_Ptr : Char_Star := Source_Ptr;
      Element : C.char;
   begin
      if Target_Temp_Ptr = null or Source_Temp_Ptr = null then
         raise C.Strings.Dereference_Error;
      end if;
```

49/1
```
{8652/0065} {AI95-00142-01}       loop
            Element            := Source_Temp_Ptr.all;
            Target_Temp_Ptr.all := Element;
            exit when C."="(Element, C.nul);
            Char_Ptrs.Increment(Target_Temp_Ptr);
            Char_Ptrs.Increment(Source_Temp_Ptr);
         end loop;
      end Strcpy;
   begin
      ...
   end Test_Pointers;
```

B.3.3 Pragma Unchecked_Union

1/2 {*AI95-00216-01*} {*union (C)*} [A pragma Unchecked_Union specifies an interface correspondence between a given discriminated type and some C union. The pragma specifies that the associated type shall be given a representation that leaves no space for its discriminant(s).]

Syntax

2/2 {*AI95-00216-01*} The form of a pragma Unchecked_Union is as follows:

3/2 **pragma** Unchecked_Union (*first_subtype*_local_name);

Legality Rules

4/2 {*AI95-00216-01*} Unchecked_Union is a representation pragma, specifying the unchecked union aspect of representation.

5/2 {*AI95-00216-01*} The *first_subtype*_local_name of a pragma Unchecked_Union shall denote an unconstrained discriminated record subtype having a variant_part.

6/2 {*AI95-00216-01*} {*unchecked union type*} {*unchecked union subtype*} {*unchecked union object*} A type to which a pragma Unchecked_Union applies is called an *unchecked union type*. A subtype of an unchecked union type is defined to be an *unchecked union subtype*. An object of an unchecked union type is defined to be an *unchecked union object*.

7/2 {*AI95-00216-01*} All component subtypes of an unchecked union type shall be C-compatible.

8/2 {*AI95-00216-01*} If a component subtype of an unchecked union type is subject to a per-object constraint, then the component subtype shall be an unchecked union subtype.

9/2 {*AI95-00216-01*} Any name that denotes a discriminant of an object of an unchecked union type shall occur within the declarative region of the type.

10/2 {*AI95-00216-01*} A component declared in a variant_part of an unchecked union type shall not have a controlled, protected, or task part.

11/2 {*AI95-00216-01*} The completion of an incomplete or private type declaration having a known_discriminant_part shall not be an unchecked union type.

12/2 {*AI95-00216-01*} An unchecked union subtype shall only be passed as a generic actual parameter if the corresponding formal type has no known discriminants or is an unchecked union type.

12.a/2 **Ramification:** This includes formal private types without a known_discriminant_part, formal derived types that do not inherit any discriminants (formal derived types do not have known_discriminant_parts), and formal derived types that are unchecked union types.

Static Semantics

13/2 {*AI95-00216-01*} An unchecked union type is eligible for convention C.

{*AI95-00216-01*} All objects of an unchecked union type have the same size. 14/2

{*AI95-00216-01*} Discriminants of objects of an unchecked union type are of size zero. 15/2

{*AI95-00216-01*} Any check which would require reading a discriminant of an unchecked union object is suppressed (see 11.5). These checks include: 16/2

- The check performed when addressing a variant component (i.e., a component that was declared in a variant part) of an unchecked union object that the object has this component (see 4.1.3). 17/2
- Any checks associated with a type or subtype conversion of a value of an unchecked union type (see 4.6). This includes, for example, the check associated with the implicit subtype conversion of an assignment statement. 18/2
- The subtype membership check associated with the evaluation of a qualified expression (see 4.7) or an uninitialized allocator (see 4.8). 19/2

Discussion: If a suppressed check would have failed, execution is erroneous (see 11.5). An implementation is always allowed to make a suppressed check if it can somehow determine the discriminant value. 19.a/2

Dynamic Semantics

{*AI95-00216-01*} A view of an unchecked union object (including a type conversion or function call) has *inferable discriminants* if it has a constrained nominal subtype, unless the object is a component of an enclosing unchecked union object that is subject to a per-object constraint and the enclosing object lacks inferable discriminants.{*inferable discriminants*} 20/2

{*AI95-00216-01*} An expression of an unchecked union type has inferable discriminants if it is either a name of an object with inferable discriminants or a qualified expression whose subtype_mark denotes a constrained subtype. 21/2

{*AI95-00216-01*} Program_Error is raised in the following cases:{*Program_Error (raised by failure of run-time check)*} 22/2

- Evaluation of the predefined equality operator for an unchecked union type if either of the operands lacks inferable discriminants. 23/2
- Evaluation of the predefined equality operator for a type which has a subcomponent of an unchecked union type whose nominal subtype is unconstrained. 24/2
- Evaluation of a membership test if the subtype_mark denotes a constrained unchecked union subtype and the expression lacks inferable discriminants. 25/2
- Conversion from a derived unchecked union type to an unconstrained non-unchecked-union type if the operand of the conversion lacks inferable discriminants. 26/2
- Execution of the default implementation of the Write or Read attribute of an unchecked union type. 27/2
- Execution of the default implementation of the Output or Input attribute of an unchecked union type if the type lacks default discriminant values. 28/2

Implementation Permissions

{*AI95-00216-01*} An implementation may require that pragma Controlled be specified for the type of an access subcomponent of an unchecked union type. 29/2

NOTES

30/2 13 {*AI95-00216-01*} The use of an unchecked union to obtain the effect of an unchecked conversion results in erroneous execution (see 11.5). Execution of the following example is erroneous even if Float'Size = Integer'Size:

31/2
```
type T (Flag : Boolean := False) is
   record
       case Flag is
           when False =>
               F1 : Float := 0.0;
           when True =>
               F2 : Integer := 0;
       end case;
    end record;
pragma Unchecked_Union (T);
```

32/2
```
X : T;
Y : Integer := X.F2; -- erroneous
```

Extensions to Ada 95

32.a/2 {*AI95-00216-01*} {*extensions to Ada 95*} Pragma Unchecked_Union is new.

B.4 Interfacing with COBOL

1 {*interface to COBOL*} {*COBOL interface*} The facilities relevant to interfacing with the COBOL language are the package Interfaces.COBOL and support for the Import, Export and Convention pragmas with *convention_*identifier COBOL.

2 The COBOL interface package supplies several sets of facilities:

3 • A set of types corresponding to the native COBOL types of the supported COBOL implementation (so-called “internal COBOL representations”), allowing Ada data to be passed as parameters to COBOL programs

4 • A set of types and constants reflecting external data representations such as might be found in files or databases, allowing COBOL-generated data to be read by an Ada program, and Ada-generated data to be read by COBOL programs

5 • A generic package for converting between an Ada decimal type value and either an internal or external COBOL representation

Static Semantics

6 The library package Interfaces.COBOL has the following declaration:

7
```
package Interfaces.COBOL is
   pragma Preelaborate(COBOL);
```

8
```
-- Types and operations for internal data representations
```

9
```
   type Floating      is digits implementation-defined;
   type Long_Floating is digits implementation-defined;
```

10
```
   type Binary      is range implementation-defined;
   type Long_Binary is range implementation-defined;
```

11
```
   Max_Digits_Binary      : constant := implementation-defined;
   Max_Digits_Long_Binary : constant := implementation-defined;
```

12
```
   type Decimal_Element  is mod implementation-defined;
   type Packed_Decimal is array (Positive range <>) of Decimal_Element;
   pragma Pack(Packed_Decimal);
```

13
```
   type COBOL_Character is implementation-defined character type;
```

14
```
   Ada_To_COBOL : array (Character) of COBOL_Character := implementation-defined;
```

15
```
   COBOL_To_Ada : array (COBOL_Character) of Character := implementation-defined;
```

16
```
   type Alphanumeric is array (Positive range <>) of COBOL_Character;
   pragma Pack(Alphanumeric);
   function To_COBOL (Item : in String) return Alphanumeric;                    17
   function To_Ada   (Item : in Alphanumeric) return String;

   procedure To_COBOL (Item       : in String;                                  18
                       Target     : out Alphanumeric;
                       Last       : out Natural);

   procedure To_Ada (Item     : in Alphanumeric;                                19
                     Target   : out String;
                     Last     : out Natural);

   type Numeric is array (Positive range <>) of COBOL_Character;                20
   pragma Pack(Numeric);

-- Formats for COBOL data representations                                       21

   type Display_Format is private;                                              22

   Unsigned             : constant Display_Format;                              23
   Leading_Separate     : constant Display_Format;
   Trailing_Separate    : constant Display_Format;
   Leading_Nonseparate  : constant Display_Format;
   Trailing_Nonseparate : constant Display_Format;

   type Binary_Format is private;                                               24

   High_Order_First  : constant Binary_Format;                                  25
   Low_Order_First   : constant Binary_Format;
   Native_Binary     : constant Binary_Format;

   type Packed_Format is private;                                               26

   Packed_Unsigned   : constant Packed_Format;                                  27
   Packed_Signed     : constant Packed_Format;

-- Types for external representation of COBOL binary data                       28

   type Byte is mod 2**COBOL_Character'Size;                                    29
   type Byte_Array is array (Positive range <>) of Byte;
   pragma Pack (Byte_Array);

   Conversion_Error : exception;                                                30

   generic                                                                      31
      type Num is delta <> digits <>;
   package Decimal_Conversions is

      -- Display Formats: data values are represented as Numeric                32

      function Valid (Item   : in Numeric;                                      33
                      Format : in Display_Format) return Boolean;

      function Length (Format : in Display_Format) return Natural;              34

      function To_Decimal (Item   : in Numeric;                                 35
                           Format : in Display_Format) return Num;

      function To_Display (Item   : in Num;                                     36
                           Format : in Display_Format) return Numeric;

      -- Packed Formats: data values are represented as Packed_Decimal          37

      function Valid (Item   : in Packed_Decimal;                               38
                      Format : in Packed_Format) return Boolean;

      function Length (Format : in Packed_Format) return Natural;               39

      function To_Decimal (Item   : in Packed_Decimal;                          40
                           Format : in Packed_Format) return Num;

      function To_Packed (Item   : in Num;                                      41
                          Format : in Packed_Format) return Packed_Decimal;

      -- Binary Formats: external data values are represented as Byte_Array     42

      function Valid (Item   : in Byte_Array;                                   43
                      Format : in Binary_Format) return Boolean;

      function Length (Format : in Binary_Format) return Natural;               44
      function To_Decimal (Item   : in Byte_Array;
                           Format : in Binary_Format) return Num;
```

45
```
         function To_Binary (Item   : in Num;
                             Format : in Binary_Format) return Byte_Array;
```

46
```
         -- Internal Binary formats: data values are of type Binary or Long_Binary
```

47
```
         function To_Decimal (Item : in Binary)      return Num;
         function To_Decimal (Item : in Long_Binary) return Num;
```

48
```
         function To_Binary      (Item : in Num)  return Binary;
         function To_Long_Binary (Item : in Num)  return Long_Binary;
```

49
```
      end Decimal_Conversions;
```

50
```
   private
      ... -- not specified by the language
   end Interfaces.COBOL;
```

50.a/1 **Implementation defined:** The types Floating, Long_Floating, Binary, Long_Binary, Decimal_Element, and COBOL_Character; and the initializations of the variables Ada_To_COBOL and COBOL_To_Ada, in Interfaces.COBOL.

51 Each of the types in Interfaces.COBOL is COBOL-compatible.

52 The types Floating and Long_Floating correspond to the native types in COBOL for data items with computational usage implemented by floating point. The types Binary and Long_Binary correspond to the native types in COBOL for data items with binary usage, or with computational usage implemented by binary.

53 Max_Digits_Binary is the largest number of decimal digits in a numeric value that is represented as Binary. Max_Digits_Long_Binary is the largest number of decimal digits in a numeric value that is represented as Long_Binary.

54 The type Packed_Decimal corresponds to COBOL's packed-decimal usage.

55 The type COBOL_Character defines the run-time character set used in the COBOL implementation. Ada_To_COBOL and COBOL_To_Ada are the mappings between the Ada and COBOL run-time character sets.

55.a **Reason:** The character mappings are visible variables, since the user needs the ability to modify them at run time.

56 Type Alphanumeric corresponds to COBOL's alphanumeric data category.

57 Each of the functions To_COBOL and To_Ada converts its parameter based on the mappings Ada_To_COBOL and COBOL_To_Ada, respectively. The length of the result for each is the length of the parameter, and the lower bound of the result is 1. Each component of the result is obtained by applying the relevant mapping to the corresponding component of the parameter.

58 Each of the procedures To_COBOL and To_Ada copies converted elements from Item to Target, using the appropriate mapping (Ada_To_COBOL or COBOL_To_Ada, respectively). The index in Target of the last element assigned is returned in Last (0 if Item is a null array). {*Constraint_Error (raised by failure of run-time check)*} If Item'Length exceeds Target'Length, Constraint_Error is propagated.

59 Type Numeric corresponds to COBOL's numeric data category with display usage.

60 The types Display_Format, Binary_Format, and Packed_Format are used in conversions between Ada decimal type values and COBOL internal or external data representations. The value of the constant Native_Binary is either High_Order_First or Low_Order_First, depending on the implementation.

61
```
   function Valid (Item   : in Numeric;
                   Format : in Display_Format) return Boolean;
```

62 The function Valid checks that the Item parameter has a value consistent with the value of Format. If the value of Format is other than Unsigned, Leading_Separate, and Trailing_Separate, the effect is implementation defined. If Format does have one of these values, the following rules apply:

- {*8652/0066*} {*AI95-00071-01*} Format=Unsigned: if Item comprises one or more decimal digit characters then Valid returns True, else it returns False. 63/1
- {*8652/0066*} {*AI95-00071-01*} Format=Leading_Separate: if Item comprises a single occurrence of the plus or minus sign character, and then one or more decimal digit characters, then Valid returns True, else it returns False. 64/1
- {*8652/0066*} {*AI95-00071-01*} Format=Trailing_Separate: if Item comprises one or more decimal digit characters and finally a plus or minus sign character, then Valid returns True, else it returns False. 65/1

66
```
function Length (Format : in Display_Format) return Natural;
```

67 The Length function returns the minimal length of a Numeric value sufficient to hold any value of type Num when represented as Format.

68
```
function To_Decimal (Item   : in Numeric;
                     Format : in Display_Format) return Num;
```

69 Produces a value of type Num corresponding to Item as represented by Format. The number of digits after the assumed radix point in Item is Num'Scale. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

69.a **Discussion:** There is no issue of truncation versus rounding, since the number of decimal places is established by Num'Scale.

70
```
function To_Display (Item   : in Num;
                     Format : in Display_Format) return Numeric;
```

71/1 {*8652/0067*} {*AI95-00072-01*} This function returns the Numeric value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Num is negative and Format is Unsigned.

72
```
function Valid (Item   : in Packed_Decimal;
                Format : in Packed_Format) return Boolean;
```

73 This function returns True if Item has a value consistent with Format, and False otherwise. The rules for the formation of Packed_Decimal values are implementation defined.

74
```
function Length (Format : in Packed_Format) return Natural;
```

75 This function returns the minimal length of a Packed_Decimal value sufficient to hold any value of type Num when represented as Format.

76
```
function To_Decimal (Item   : in Packed_Decimal;
                     Format : in Packed_Format) return Num;
```

77 Produces a value of type Num corresponding to Item as represented by Format. Num'Scale is the number of digits after the assumed radix point in Item. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

78
```
function To_Packed (Item   : in Num;
                    Format : in Packed_Format) return Packed_Decimal;
```

79/1 {*8652/0067*} {*AI95-00072-01*} This function returns the Packed_Decimal value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Num is negative and Format is Packed_Unsigned.

80
```
function Valid (Item   : in Byte_Array;
                Format : in Binary_Format) return Boolean;
```

81 This function returns True if Item has a value consistent with Format, and False otherwise.

81.a **Ramification:** This function returns False only when the represented value is outside the range of Num.

82
```
function Length (Format : in Binary_Format) return Natural;
```

83 This function returns the minimal length of a Byte_Array value sufficient to hold any value of type Num when represented as Format.

84
```
function To_Decimal (Item   : in Byte_Array;
                     Format : in Binary_Format) return Num;
```

85 Produces a value of type Num corresponding to Item as represented by Format. Num'Scale is the number of digits after the assumed radix point in Item. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

86
```
function To_Binary (Item   : in Num;
                    Format : in Binary_Format) return Byte_Array;
```

87/1 {*8652/0067*} {*AI95-00072-01*} This function returns the Byte_Array value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1.

88
```
function To_Decimal (Item : in Binary)      return Num;

function To_Decimal (Item : in Long_Binary) return Num;
```

89 These functions convert from COBOL binary format to a corresponding value of the decimal type Num. Conversion_Error is propagated if Item is too large for Num.

89.a **Ramification:** There is no rescaling performed on the conversion. That is, the returned value in each case is a “bit copy” if Num has a binary radix. The programmer is responsible for maintaining the correct scale.

90
```
function To_Binary      (Item : in Num)  return Binary;

function To_Long_Binary (Item : in Num)  return Long_Binary;
```

91 These functions convert from Ada decimal to COBOL binary format. Conversion_Error is propagated if the value of Item is too large to be represented in the result type.

91.a **Discussion:** One style of interface supported for COBOL, similar to what is provided for C, is the ability to call and pass parameters to an existing COBOL program. Thus the interface package supplies types that can be used in an Ada program as parameters to subprograms whose bodies will be in COBOL. These types map to COBOL's alphanumeric and numeric data categories.

91.b Several types are provided for support of alphanumeric data. Since COBOL's run-time character set is not necessarily the same as Ada's, Interfaces.COBOL declares an implementation-defined character type COBOL_Character, and mappings between Character and COBOL_Character. These mappings are visible variables (rather than, say, functions or constant arrays), since in the situation where COBOL_Character is EBCDIC, the flexibility of dynamically modifying the mappings is needed. Corresponding to COBOL's alphanumeric data is the string type Alphanumeric.

91.c Numeric data may have either a “display” or “computational” representation in COBOL. On the Ada side, the data is of a decimal fixed point type. Passing an Ada decimal data item to a COBOL program requires conversion from the Ada decimal type to some type that reflects the representation expected on the COBOL side.

91.d
- Computational Representation

91.e Floating point representation is modeled by Ada floating point types, Floating and Long_Floating. Conversion between these types and Ada decimal types is obtained directly, since the type name serves as a conversion function.

91.f Binary representation is modeled by an Ada integer type, Binary, and possibly other types such as Long_Binary. Conversion between, say, Binary and a decimal type is through functions from an instantiation of the generic package Decimal_Conversions.

91.g Packed decimal representation is modeled by the Ada array type Packed_Decimal. Conversion between packed decimal and a decimal type is through functions from an instantiation of the generic package Decimal_Conversions.

91.h
- Display Representation

91.i Display representation for numeric data is modeled by the array type Numeric. Conversion between display representation and a decimal type is through functions from an instantiation of the generic package

Decimal_Conversions. A parameter to the conversion function indicates the desired interpretation of the data (e.g., signed leading separate, etc.)

91.j Pragma Convention(COBOL, T) may be applied to a record type T to direct the compiler to choose a COBOL-compatible representation for objects of the type.

91.k The package Interfaces.COBOL allows the Ada programmer to deal with data from files (or databases) created by a COBOL program. For data that is alphanumeric, or in display or packed decimal format, the approach is the same as for passing parameters (instantiate Decimal_Conversions to obtain the needed conversion functions). For binary data, the external representation is treated as a Byte array, and an instantiation of Decimal_IO produces a package that declares the needed conversion functions. A parameter to the conversion function indicates the desired interpretation of the data (e.g., high- versus low-order byte first).

Implementation Requirements

92 An implementation shall support pragma Convention with a COBOL *convention_*identifier for a COBOL-eligible type (see B.1).

92.a **Ramification:** An implementation supporting this package shall ensure that if the bounds of a Packed_Decimal, Alphanumeric, or Numeric variable are static, then the representation of the object comprises solely the array components (that is, there is no implicit run-time “descriptor” that is part of the object).

Implementation Permissions

93 An implementation may provide additional constants of the private types Display_Format, Binary_Format, or Packed_Format.

93.a **Reason:** This is to allow exploitation of other external formats that may be available in the COBOL implementation.

94 An implementation may provide further floating point and integer types in Interfaces.COBOL to match additional native COBOL types, and may also supply corresponding conversion functions in the generic package Decimal_Conversions.

Implementation Advice

95 An Ada implementation should support the following interface correspondences between Ada and COBOL.

- 96 An Ada **access** T parameter is passed as a “BY REFERENCE” data item of the COBOL type corresponding to T.
- 97 An Ada **in** scalar parameter is passed as a “BY CONTENT” data item of the corresponding COBOL type.
- 98 Any other Ada parameter is passed as a “BY REFERENCE” data item of the COBOL type corresponding to the Ada parameter type; for scalars, a local copy is used if necessary to ensure by-copy semantics.

98.a/2 **Implementation Advice:** If COBOL interfacing is supported, the interface correspondences between Ada and COBOL should be supported.

NOTES

99 14 An implementation is not required to support pragma Convention for access types, nor is it required to support pragma Import, Export or Convention for functions.

99.a **Reason:** COBOL does not have a pointer facility, and a COBOL program does not return a value.

100 15 If an Ada subprogram is exported to COBOL, then a call from COBOL call may specify either “BY CONTENT” or “BY REFERENCE”.

Examples

101 *Examples of Interfaces.COBOL:*

102
```
with Interfaces.COBOL;
procedure Test_Call is
```

103
```
      -- Calling a foreign COBOL program
      -- Assume that a COBOL program PROG has the following declaration
      --  in its LINKAGE section:
      --  01 Parameter-Area
      --     05 NAME   PIC X(20).
      --     05 SSN    PIC X(9).
      --     05 SALARY PIC 99999V99 USAGE COMP.
      -- The effect of PROG is to update SALARY based on some algorithm
```
104
```
      package COBOL renames Interfaces.COBOL;
```
105
```
      type Salary_Type is delta 0.01 digits 7;
```
106
```
      type COBOL_Record is
         record
            Name   : COBOL.Numeric(1..20);
            SSN    : COBOL.Numeric(1..9);
            Salary : COBOL.Binary;  -- Assume Binary = 32 bits
         end record;
      pragma Convention (COBOL, COBOL_Record);
```
107
```
      procedure Prog (Item : in out COBOL_Record);
      pragma Import (COBOL, Prog, "PROG");
```
108
```
      package Salary_Conversions is
         new COBOL.Decimal_Conversions(Salary_Type);
```
109
```
      Some_Salary : Salary_Type := 12_345.67;
      Some_Record : COBOL_Record :=
         (Name   => "Johnson, John       ",
          SSN    => "111223333",
          Salary => Salary_Conversions.To_Binary(Some_Salary));
```
110
```
   begin
      Prog (Some_Record);
      ...
   end Test_Call;
```
111
```
   with Interfaces.COBOL;
   with COBOL_Sequential_IO; -- Assumed to be supplied by implementation
   procedure Test_External_Formats is
```
112
```
      -- Using data created by a COBOL program
      -- Assume that a COBOL program has created a sequential file with
      --  the following record structure, and that we need to
      --  process the records in an Ada program
      --  01 EMPLOYEE-RECORD
      --     05 NAME    PIC X(20).
      --     05 SSN     PIC X(9).
      --     05 SALARY  PIC 99999V99 USAGE COMP.
      --     05 ADJUST  PIC S999V999 SIGN LEADING SEPARATE.
      -- The COMP data is binary (32 bits), high-order byte first
```
113
```
      package COBOL renames Interfaces.COBOL;
```
114
```
      type Salary_Type      is delta 0.01  digits 7;
      type Adjustments_Type is delta 0.001 digits 6;
```
115
```
      type COBOL_Employee_Record_Type is  -- External representation
         record
            Name    : COBOL.Alphanumeric(1..20);
            SSN     : COBOL.Alphanumeric(1..9);
            Salary  : COBOL.Byte_Array(1..4);
            Adjust  : COBOL.Numeric(1..7);  -- Sign and 6 digits
         end record;
      pragma Convention (COBOL, COBOL_Employee_Record_Type);
```
116
```
      package COBOL_Employee_IO is
         new COBOL_Sequential_IO(COBOL_Employee_Record_Type);
      use COBOL_Employee_IO;
```
117
```
      COBOL_File : File_Type;
```

118
```
      type Ada_Employee_Record_Type is  -- Internal representation
         record
            Name    : String(1..20);
            SSN     : String(1..9);
            Salary  : Salary_Type;
            Adjust  : Adjustments_Type;
         end record;
```

119
```
      COBOL_Record : COBOL_Employee_Record_Type;
      Ada_Record   : Ada_Employee_Record_Type;
```

120
```
      package Salary_Conversions is
         new COBOL.Decimal_Conversions(Salary_Type);
      use Salary_Conversions;
```

121
```
      package Adjustments_Conversions is
         new COBOL.Decimal_Conversions(Adjustments_Type);
      use Adjustments_Conversions;
```

122
```
   begin
      Open (COBOL_File, Name => "Some_File");
```

123
```
      loop
        Read (COBOL_File, COBOL_Record);
```

124
```
        Ada_Record.Name := To_Ada(COBOL_Record.Name);
        Ada_Record.SSN  := To_Ada(COBOL_Record.SSN);
        Ada_Record.Salary :=
           To_Decimal(COBOL_Record.Salary, COBOL.High_Order_First);
        Ada_Record.Adjust :=
           To_Decimal(COBOL_Record.Adjust, COBOL.Leading_Separate);
        ... -- Process Ada_Record
      end loop;
   exception
      when End_Error => ...
   end Test_External_Formats;
```

Wording Changes from Ada 95

124.a/2 {*8652/0066*} {*AI95-00071-01*} **Corrigendum:** Corrected the definition of Valid to match COBOL.

124.b/2 {*8652/0067*} {*AI95-00072-01*} **Corrigendum:** Specified the bounds of the results of To_Display, To_Packed, and To_Binary.

B.5 Interfacing with Fortran

1 {*interface to Fortran*} {*Fortran interface*} The facilities relevant to interfacing with the Fortran language are the package Interfaces.Fortran and support for the Import, Export and Convention pragmas with *convention_*identifier Fortran.

2 The package Interfaces.Fortran defines Ada types whose representations are identical to the default representations of the Fortran intrinsic types Integer, Real, Double Precision, Complex, Logical, and Character in a supported Fortran implementation. These Ada types can therefore be used to pass objects between Ada and Fortran programs.

Static Semantics

3 The library package Interfaces.Fortran has the following declaration:

4
```
   with Ada.Numerics.Generic_Complex_Types;  -- see G.1.1
   pragma Elaborate_All(Ada.Numerics.Generic_Complex_Types);
   package Interfaces.Fortran is
      pragma Pure(Fortran);
```

5
```
      type Fortran_Integer is range implementation-defined;
```

6
```
      type Real             is digits implementation-defined;
      type Double_Precision is digits implementation-defined;
```

7
```
      type Logical is new Boolean;
```

8
```
      package Single_Precision_Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (Real);
```
9
```
      type Complex is new Single_Precision_Complex_Types.Complex;
```
10
```
      subtype Imaginary is Single_Precision_Complex_Types.Imaginary;
      i : Imaginary renames Single_Precision_Complex_Types.i;
      j : Imaginary renames Single_Precision_Complex_Types.j;
```
11
```
      type Character_Set is implementation-defined character type;
```
12
```
      type Fortran_Character is array (Positive range <>) of Character_Set;
      pragma Pack (Fortran_Character);
```
13
```
      function To_Fortran (Item : in Character) return Character_Set;
      function To_Ada (Item : in Character_Set) return Character;
```
14
```
      function To_Fortran (Item : in String) return Fortran_Character;
      function To_Ada     (Item : in Fortran_Character) return String;
```
15
```
      procedure To_Fortran (Item       : in String;
                            Target     : out Fortran_Character;
                            Last       : out Natural);
```
16
```
      procedure To_Ada (Item     : in Fortran_Character;
                        Target   : out String;
                        Last     : out Natural);
```
17
```
   end Interfaces.Fortran;
```

17.a.1/1 **Implementation defined:** The types Fortran_Integer, Real, Double_Precision, and Character_Set in Interfaces.Fortran.

17.a **Ramification:** The means by which the Complex type is provided in Interfaces.Fortran creates a dependence of Interfaces.Fortran on Numerics.Generic_Complex_Types (see G.1.1). This dependence is intentional and unavoidable, if the Fortran-compatible Complex type is to be useful in Ada code without duplicating facilities defined elsewhere.

18 The types Fortran_Integer, Real, Double_Precision, Logical, Complex, and Fortran_Character are Fortran-compatible.

19 The To_Fortran and To_Ada functions map between the Ada type Character and the Fortran type Character_Set, and also between the Ada type String and the Fortran type Fortran_Character. The To_Fortran and To_Ada procedures have analogous effects to the string conversion subprograms found in Interfaces.COBOL.

Implementation Requirements

20 An implementation shall support pragma Convention with a Fortran *convention_*identifier for a Fortran-eligible type (see B.1).

Implementation Permissions

21 An implementation may add additional declarations to the Fortran interface packages. For example, the Fortran interface package for an implementation of Fortran 77 (ANSI X3.9-1978) that defines types like Integer**n*, Real**n*, Logical**n*, and Complex**n* may contain the declarations of types named Integer_Star_*n*, Real_Star_*n*, Logical_Star_*n*, and Complex_Star_*n*. (This convention should not apply to Character**n*, for which the Ada analog is the constrained array subtype Fortran_Character (1..*n*).) Similarly, the Fortran interface package for an implementation of Fortran 90 that provides multiple *kinds* of intrinsic types, e.g. Integer (Kind=*n*), Real (Kind=*n*), Logical (Kind=*n*), Complex (Kind=*n*), and Character (Kind=*n*), may contain the declarations of types with the recommended names Integer_Kind_*n*, Real_Kind_*n*, Logical_Kind_*n*, Complex_Kind_*n*, and Character_Kind_*n*.

21.a **Discussion:** Implementations may add auxiliary declarations as needed to assist in the declarations of additional Fortran-compatible types. For example, if a double precision complex type is defined, then Numerics.Generic_Complex_Types may be instantiated for the double precision type. Similarly, if a wide character type is defined to match a Fortran 90 wide character type (accessible in Fortran 90 with the Kind modifier), then an auxiliary character set may be declared to serve as its component type.

Implementation Advice

An Ada implementation should support the following interface correspondences between Ada and Fortran: 22

- An Ada procedure corresponds to a Fortran subroutine. 23
- An Ada function corresponds to a Fortran function. 24
- An Ada parameter of an elementary, array, or record type T is passed as a T_F argument to a Fortran procedure, where T_F is the Fortran type corresponding to the Ada type T, and where the INTENT attribute of the corresponding dummy argument matches the Ada formal parameter mode; the Fortran implementation's parameter passing conventions are used. For elementary types, a local copy is used if necessary to ensure by-copy semantics. 25
- An Ada parameter of an access-to-subprogram type is passed as a reference to a Fortran procedure whose interface corresponds to the designated subprogram's specification. 26

Implementation Advice: If Fortran interfacing is supported, the interface correspondences between Ada and Fortran should be supported. 26.a/2

NOTES

16 An object of a Fortran-compatible record type, declared in a library package or subprogram, can correspond to a Fortran common block; the type also corresponds to a Fortran “derived type”. 27

Examples

Example of Interfaces.Fortran: 28

29
```
with Interfaces.Fortran;
use Interfaces.Fortran;
procedure Ada_Application is
```
30
```
   type Fortran_Matrix is array (Integer range <>,
                                 Integer range <>) of Double_Precision;
   pragma Convention (Fortran, Fortran_Matrix);     -- stored in Fortran's
                                                    -- column-major order
   procedure Invert (Rank : in Fortran_Integer; X : in out Fortran_Matrix);
   pragma Import (Fortran, Invert);                 -- a Fortran subroutine
```
31
```
   Rank      : constant Fortran_Integer := 100;
   My_Matrix : Fortran_Matrix (1 .. Rank, 1 .. Rank);
```
32
```
begin
```
33
```
   ...
   My_Matrix := ...;
   ...
   Invert (Rank, My_Matrix);
   ...
```
34
```
end Ada_Application;
```

Annex C (normative) Systems Programming

1 [{*systems programming*} {*low-level programming*} {*real-time systems*} {*embedded systems*} {*distributed systems*} {*information systems*} The Systems Programming Annex specifies additional capabilities provided for low-level programming. These capabilities are also required in many real-time, embedded, distributed, and information systems.]

Extensions to Ada 83

1.a {*extensions to Ada 83*} This Annex is new to Ada 95.

C.1 Access to Machine Operations

1 [This clause specifies rules regarding access to machine instructions from within an Ada program.]

1.a/2 **Implementation defined:** Implementation-defined intrinsic subprograms.

Implementation Requirements

2 {*machine code insertion*} The implementation shall support machine code insertions (see 13.8) or intrinsic subprograms (see 6.3.1) (or both). Implementation-defined attributes shall be provided to allow the use of Ada entities as operands.

Implementation Advice

3 The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment, including privileged instructions, if any.

3.a.1/2 **Implementation Advice:** The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment.

3.a **Ramification:** Of course, on a machine with protection, an attempt to execute a privileged instruction in user mode will probably trap. Nonetheless, we want implementations to provide access to them so that Ada can be used to write systems programs that run in privileged mode.

4 {*interface to assembly language*} {*language (interface to assembly)*} {*mixed-language programs*} {*assembly language*} The interfacing pragmas (see Annex B) should support interface to assembler; the default assembler should be associated with the convention identifier Assembler.

4.a/2 **Implementation Advice:** Interface to assembler should be supported; the default assembler should be associated with the convention identifier Assembler.

5 If an entity is exported to assembly language, then the implementation should allocate it at an addressable location, and should ensure that it is retained by the linking process, even if not otherwise referenced from the Ada code. The implementation should assume that any call to a machine code or assembler subprogram is allowed to read or update every object that is specified as exported.

5.a/2 **Implementation Advice:** If an entity is exported to assembly language, then the implementation should allocate it at an addressable location even if not otherwise referenced from the Ada code. A call to a machine code or assembler subprogram should be treated as if it could read or update every object that is specified as exported.

Documentation Requirements

6 The implementation shall document the overhead associated with calling machine-code or intrinsic subprograms, as compared to a fully-inlined call, and to a regular out-of-line call.

6.a/2 **Documentation Requirement:** The overhead of calling machine-code or intrinsic subprograms.

7 The implementation shall document the types of the package System.Machine_Code usable for machine code insertions, and the attributes to be used in machine code insertions for references to Ada entities.

7.a/2 **Documentation Requirement:** The types and attributes used in machine code insertions.

8 The implementation shall document the subprogram calling conventions associated with the convention identifiers available for use with the interfacing pragmas (Ada and Assembler, at a minimum), including register saving, exception propagation, parameter passing, and function value returning.

8.a/2 **Documentation Requirement:** The subprogram calling conventions for all supported convention identifiers.

9 For exported and imported subprograms, the implementation shall document the mapping between the Link_Name string, if specified, or the Ada designator, if not, and the external link name used for such a subprogram.

9.a/2 *This paragraph was deleted.*

9.b/2 **Documentation Requirement:** The mapping between the Link_Name or Ada designator and the external link name.

Implementation Advice

10 The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms.

10.a/2 **Implementation Advice:** Little or no overhead should be associated with calling intrinsic and machine-code subprograms.

11 It is recommended that intrinsic subprograms be provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs. Examples of such instructions include:

12 • Atomic read-modify-write operations — e.g., test and set, compare and swap, decrement and test, enqueue/dequeue.

13 • Standard numeric functions — e.g., *sin*, *log*.

14 • String manipulation operations — e.g., translate and test.

15 • Vector operations — e.g., compare vector against thresholds.

16 • Direct operations on I/O ports.

16.a/2 **Implementation Advice:** Intrinsic subprograms should be provided to access any machine operations that provide special capabilities or efficiency not normally available.

C.2 Required Representation Support

1/2 {*AI95-00434-01*} This clause specifies minimal requirements on the support for representation items and related features.

Implementation Requirements

2 {*recommended level of support (required in Systems Programming Annex)* [partial]} The implementation shall support at least the functionality defined by the recommended levels of support in Section 13.

C.3 Interrupt Support

1 [This clause specifies the language-defined model for hardware interrupts in addition to mechanisms for handling interrupts.] {*signal: See interrupt*}

Dynamic Semantics

2 {*interrupt*} [An *interrupt* represents a class of events that are detected by the hardware or the system software.] {*occurrence (of an interrupt)*} Interrupts are said to occur. An *occurrence* of an interrupt is separable into generation and delivery. {*generation (of an interrupt)*} *Generation* of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program. {*delivery (of an interrupt)*} *Delivery* is the action that invokes part of the program as response to the interrupt occurrence. {*pending interrupt occurrence*} Between generation and delivery, the interrupt occurrence [(or interrupt)] is *pending*. {*blocked interrupt*} Some or all interrupts may be *blocked*. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. {*attaching (to an interrupt)*} {*reserved interrupt*} Certain interrupts are *reserved*. The set of reserved interrupts is implementation defined. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other implementation-defined means. {*interrupt handler*} {*handler (interrupt)* [partial]} Program units can be connected to non-reserved interrupts. While connected, the program unit is said to be *attached* to that interrupt. The execution of that program unit, the *interrupt handler*, is invoked upon delivery of the interrupt occurrence.

2.a/2 *This paragraph was deleted.*

2.b **To be honest:** As an obsolescent feature, interrupts may be attached to task entries by an address clause. See J.7.1.

3 While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt. While the handler executes, the corresponding interrupt is blocked.

4 While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Whether such occurrences remain pending or are lost is implementation defined.

5 {*default treatment*} Each interrupt has a *default treatment* which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is implementation defined, as is the method (if one exists) for configuring the default treatments for interrupts.

6 An interrupt is delivered to the handler (or default treatment) that is in effect for that interrupt at the time of delivery.

7 An exception propagated from a handler that is invoked by an interrupt has no effect.

8 [If the Ceiling_Locking policy (see D.3) is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object.]

Implementation Requirements

9 The implementation shall provide a mechanism to determine the minimum stack space that is needed for each interrupt handler and to reserve that space for the execution of the handler. [This space should accommodate nested invocations of the handler where the system permits this.]

10 If the hardware or the underlying system holds pending interrupt occurrences, the implementation shall provide for later delivery of these occurrences to the program.

11 If the Ceiling_Locking policy is not in effect, the implementation shall provide means for the application to specify whether interrupts are to be blocked during protected actions.

Documentation Requirements

12 The implementation shall document the following items:

12.a **Discussion:** This information may be different for different forms of interrupt handlers.

13 1. For each interrupt, which interrupts are blocked from delivery when a handler attached to that interrupt executes (either as a result of an interrupt delivery or of an ordinary call on a procedure of the corresponding protected object).

14 2. Any interrupts that cannot be blocked, and the effect of attaching handlers to such interrupts, if this is permitted.

15 3. Which run-time stack an interrupt handler uses when it executes as a result of an interrupt delivery; if this is configurable, what is the mechanism to do so; how to specify how much space to reserve on that stack.

16 4. Any implementation- or hardware-specific activity that happens before a user-defined interrupt handler gets control (e.g., reading device registers, acknowledging devices).

17 5. Any timing or other limitations imposed on the execution of interrupt handlers.

18 6. The state (blocked/unblocked) of the non-reserved interrupts when the program starts; if some interrupts are unblocked, what is the mechanism a program can use to protect itself before it can attach the corresponding handlers.

19 7. Whether the interrupted task is allowed to resume execution before the interrupt handler returns.

20 8. The treatment of interrupt occurrences that are generated while the interrupt is blocked; i.e., whether one or more occurrences are held for later delivery, or all are lost.

21 9. Whether predefined or implementation-defined exceptions are raised as a result of the occurrence of any interrupt, and the mapping between the machine interrupts (or traps) and the predefined exceptions.

22 10.
On a multi-processor, the rules governing the delivery of an interrupt to a particular processor.

22.a/2 **Documentation Requirement:** The treatment of interrupts.

Implementation Permissions

23/2 {*AI95-00434-01*} If the underlying system or hardware does not allow interrupts to be blocked, then no blocking is required [as part of the execution of subprograms of a protected object for which one of its subprograms is an interrupt handler].

24 In a multi-processor with more than one interrupt subsystem, it is implementation defined whether (and how) interrupt sources from separate subsystems share the same Interrupt_ID type (see C.3.2). In particular, the meaning of a blocked or pending interrupt may then be applicable to one processor only.

24.a **Discussion:** This issue is tightly related to the issue of scheduling on a multi-processor. In a sense, if a particular interrupt source is not available to all processors, the system is not truly homogeneous.

24.b One way to approach this problem is to assign sub-ranges within Interrupt_ID to each interrupt subsystem, such that "similar" interrupt sources (e.g. a timer) in different subsystems get a distinct id.

25 Implementations are allowed to impose timing or other limitations on the execution of interrupt handlers.

25.a **Reason:** These limitations are often necessary to ensure proper behavior of the implementation.

26/2 {*AI95-00434-01*} Other forms of handlers are allowed to be supported, in which case the rules of this clause should be adhered to.

27 The active priority of the execution of an interrupt handler is allowed to vary from one occurrence of the same interrupt to another.

Implementation Advice

{*AI95-00434-01*} If the Ceiling_Locking policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for finer-grained control of interrupt blocking. 28/2

Implementation Advice: If the Ceiling_Locking policy is not in effect and the target system allows for finer-grained control of interrupt blocking, a means for the application to specify which interrupts are to be blocked during protected actions should be provided. 28.a/2

NOTES

1 The default treatment for an interrupt can be to keep the interrupt pending or to deliver it to an implementation-defined handler. Examples of actions that an implementation-defined handler is allowed to perform include aborting the partition, ignoring (i.e., discarding occurrences of) the interrupt, or queuing one or more occurrences of the interrupt for possible later delivery when a user-defined handler is attached to that interrupt. 29

2 It is a bounded error to call Task_Identification.Current_Task (see C.7.1) from an interrupt handler. 30

3 The rule that an exception propagated from an interrupt handler has no effect is modeled after the rule about exceptions propagated out of task bodies. 31

C.3.1 Protected Procedure Handlers

Syntax

The form of a pragma Interrupt_Handler is as follows: 1

pragma Interrupt_Handler(*handler*_name); 2

The form of a pragma Attach_Handler is as follows: 3

pragma Attach_Handler(*handler*_name, expression); 4

Name Resolution Rules

For the Interrupt_Handler and Attach_Handler pragmas, the *handler*_name shall resolve to denote a protected procedure with a parameterless profile. 5

For the Attach_Handler pragma, the expected type for the expression is Interrupts.Interrupt_ID (see C.3.2). 6

Legality Rules

{*AI95-00434-01*} The Attach_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_-type_declaration or single_protected_declaration shall be a library-level declaration. 7/2

Discussion: In the case of a protected_type_declaration, an object_declaration of an object of that type need not be at library level. 7.a

{*AI95-00253-01*} {*AI95-00303-01*} The Interrupt_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration. 8/2

Dynamic Semantics

If the pragma Interrupt_Handler appears in a protected_definition, then the corresponding procedure can be attached dynamically, as a handler, to interrupts (see C.3.2). [Such procedures are allowed to be attached to multiple interrupts.] 9

{*creation (of a protected object)*} {*initialization (of a protected object)*} The expression in the Attach_Handler pragma [as evaluated at object creation time] specifies an interrupt. As part of the initialization of that object, if the Attach_Handler pragma is specified, the *handler* procedure is attached to the specified interrupt. {*Reserved_Check* [partial]} {*check, language-defined (Reserved_Check)*} A check is made that the 10 corresponding interrupt is not reserved. {*Program_Error (raised by failure of run-time check)*} Program_Error is raised if the check fails, and the existing treatment for the interrupt is not affected.

11/2 {*AI95-00434-01*} {*initialization (of a protected object)*} {*Ceiling_Check* [partial]} {*check, language-defined (Ceiling_Check)*} If the Ceiling_Locking policy (see D.3) is in effect, then upon the initialization of a protected object for which either an Attach_Handler or Interrupt_Handler pragma applies to one of its procedures, a check is made that the ceiling priority defined in the protected_definition is in the range of System.Interrupt_Priority. {*Program_Error (raised by failure of run-time check)*} If the check fails, Program_Error is raised.

12/1 {*8652/0068*} {*AI95-00121-01*} {*finalization (of a protected object)*} When a protected object is finalized, for any of its procedures that are attached to interrupts, the handler is detached. If the handler was attached by a procedure in the Interrupts package or if no user handler was previously attached to the interrupt, the default treatment is restored. If an Attach_Handler pragma was used and the most recently attached handler for the same interrupt is the same as the one that was attached at the time the protected object was initialized, the previous handler is restored.

12.a/2 **Discussion:** {*8652/0068*} {*AI95-00121-01*} {*AI95-00303-01*} If all protected objects for interrupt handlers are declared at the library level, the finalization discussed above occurs only as part of the finalization of all library-level packages in a partition. However, objects of a protected type containing an Attach_Handler pragma need not be at the library level. Thus, an implementation needs to be able to restore handlers during the execution of the program. (An object with an Interrupt_Handler pragma also need not be at the library level, but such a handler cannot be attached to an interrupt using the Interrupts package.)

13 When a handler is attached to an interrupt, the interrupt is blocked [(subject to the Implementation Permission in C.3)] during the execution of every protected action on the protected object containing the handler.

Erroneous Execution

14 {*erroneous execution (cause)* [partial]} If the Ceiling_Locking policy (see D.3) is in effect and an interrupt is delivered to a handler, and the interrupt hardware priority is higher than the ceiling priority of the corresponding protected object, the execution of the program is erroneous.

14.1/1 {*8652/0068*} {*AI95-00121-01*} {*erroneous execution (cause)* [partial]} If the handlers for a given interrupt attached via pragma Attach_Handler are not attached and detached in a stack-like (LIFO) order, program execution is erroneous. In particular, when a protected object is finalized, the execution is erroneous if any of the procedures of the protected object are attached to interrupts via pragma Attach_Handler and the most recently attached handler for the same interrupt is not the same as the one that was attached at the time the protected object was initialized.

14.a.1/1 **Discussion:** {*8652/0068*} {*AI95-00121-01*} This simplifies implementation of the Attach_Handler pragma by not requiring a check that the current handler is the same as the one attached by the initialization of a protected object.

Metrics

15 The following metric shall be documented by the implementation:

16/2
- {*AI95-00434-01*} The worst-case overhead for an interrupt handler that is a parameterless protected procedure, in clock cycles. This is the execution time not directly attributable to the handler procedure or the interrupted execution. It is estimated as C – (A+B), where A is how long it takes to complete a given sequence of instructions without any interrupt, B is how long it takes to complete a normal call to a given protected procedure, and C is how long it takes to complete the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt.

16.a **Implementation Note:** The instruction sequence and interrupt handler used to measure interrupt handling overhead should be chosen so as to maximize the execution time cost due to cache misses. For example, if the processor has cache memory and the activity of an interrupt handler could invalidate the contents of cache memory, the handler should be written such that it invalidates all of the cache memory.

Documentation Requirement: The metrics for interrupt handlers. 16.b/2

Implementation Permissions

When the pragmas Attach_Handler or Interrupt_Handler apply to a protected procedure, the implementation is allowed to impose implementation-defined restrictions on the corresponding protected_type_declaration and protected_body. 17

Ramification: The restrictions may be on the constructs that are allowed within them, and on ordinary calls (i.e. not via interrupts) on protected operations in these protected objects. 17.a

Implementation defined: Any restrictions on a protected procedure or its containing type when a pragma Attach_handler or Interrupt_Handler applies. 17.b/2

An implementation may use a different mechanism for invoking a protected procedure in response to a hardware interrupt than is used for a call to that protected procedure from a task. 18

Discussion: This is despite the fact that the priority of an interrupt handler (see D.1) is modeled after a hardware task calling the handler. 18.a

{*notwithstanding*} Notwithstanding what this subclause says elsewhere, the Attach_Handler and Interrupt_Handler pragmas are allowed to be used for other, implementation defined, forms of interrupt handlers. 19

Ramification: For example, if an implementation wishes to allow interrupt handlers to have parameters, it is allowed to do so via these pragmas; it need not invent implementation-defined pragmas for the purpose. 19.a

Implementation defined: Any other forms of interrupt handler supported by the Attach_Handler and Interrupt_Handler pragmas. 19.b/2

Implementation Advice

Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware. 20

Implementation Advice: Interrupt handlers should be called directly by the hardware. 20.a/2

Whenever practical, the implementation should detect violations of any implementation-defined restrictions before run time. 21

Implementation Advice: Violations of any implementation-defined restrictions on interrupt handlers should be detected before run time. 21.a/2

NOTES

4 The Attach_Handler pragma can provide static attachment of handlers to interrupts if the implementation supports preelaboration of protected objects. (See C.4.) 22

5 {*AI95-00434-01*} A protected object that has a (protected) procedure attached to an interrupt should have a ceiling priority at least as high as the highest processor priority at which that interrupt will ever be delivered. 23/2

6 Protected procedures can also be attached dynamically to interrupts via operations declared in the predefined package Interrupts. 24

7 An example of a possible implementation-defined restriction is disallowing the use of the standard storage pools within the body of a protected procedure that is an interrupt handler. 25

Incompatibilities With Ada 95

{*AI95-00253-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** Corrected the wording so that the rules for the use of Attach_Handler and Interrupt_Handler are identical. This means that uses of pragma Interrupt_Handler outside of the target protected type or single protected object are now illegal. 25.a/2

Wording Changes from Ada 95

{*8652/0068*} {*AI95-00121-01*} **Corrigendum:** Clarified the meaning of “the previous handler” when finalizing protected objects containing interrupt handlers. 25.b/2

25.c/2 {*AI95-00303-01*} Dropped the requirement that an object of a type containing an Interrupt_Handler pragma must be declared at the library level. This was a generic contract model violation. This change is not an extension, as an attempt to attach such a handler with a routine in package Interrupts will fail an accessibility check anyway. Moreover, implementations can retain the rule as an implementation-defined restriction on the use of the type, as permitted by the Implementation Permissions above.

C.3.2 The Package Interrupts

Static Semantics

1 The following language-defined packages exist:

2
```
with System;
package Ada.Interrupts is
   type Interrupt_ID is implementation-defined;
   type Parameterless_Handler is
      access protected procedure;
```

3/1 *This paragraph was deleted.*

4
```
   function Is_Reserved (Interrupt : Interrupt_ID)
      return Boolean;
```

5
```
   function Is_Attached (Interrupt : Interrupt_ID)
      return Boolean;
```

6
```
   function Current_Handler (Interrupt : Interrupt_ID)
      return Parameterless_Handler;
```

7
```
   procedure Attach_Handler
      (New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_ID);
```

8
```
   procedure Exchange_Handler
      (Old_Handler : out Parameterless_Handler;
       New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_ID);
```

9
```
   procedure Detach_Handler
      (Interrupt : in Interrupt_ID);
```

10
```
   function Reference(Interrupt : Interrupt_ID)
      return System.Address;
```

11
```
private
   ... -- not specified by the language
end Ada.Interrupts;
```

12
```
package Ada.Interrupts.Names is
   implementation-defined : constant Interrupt_ID :=
     implementation-defined;
      . . .
   implementation-defined : constant Interrupt_ID :=
     implementation-defined;
end Ada.Interrupts.Names;
```

Dynamic Semantics

13 The Interrupt_ID type is an implementation-defined discrete type used to identify interrupts.

14 The Is_Reserved function returns True if and only if the specified interrupt is reserved.

15 The Is_Attached function returns True if and only if a user-specified interrupt handler is attached to the interrupt.

16/1 {*8652/0069*} {*AI95-00166-01*} The Current_Handler function returns a value that represents the attached handler of the interrupt. If no user-defined handler is attached to the interrupt, Current_Handler returns **null**.

The Attach_Handler procedure attaches the specified handler to the interrupt, overriding any existing treatment (including a user handler) in effect for that interrupt. If New_Handler is **null**, the default treatment is restored. {*Program_Error (raised by failure of run-time check)*} If New_Handler designates a protected procedure to which the pragma Interrupt_Handler does not apply, Program_Error is raised. In this case, the operation does not modify the existing interrupt treatment. 17

{*8652/0069*} {*AI95-00166-01*} The Exchange_Handler procedure operates in the same manner as Attach_Handler with the addition that the value returned in Old_Handler designates the previous treatment for the specified interrupt. If the previous treatment is not a user-defined handler, **null** is returned. 18/1

Ramification: Calling Attach_Handler or Exchange_Handler with this value for New_Handler restores the previous handler. 18.a

{*8652/0069*} {*AI95-00166-01*} If the application uses only parameterless procedures as handlers (other types of handlers may be provided by the implementation, but are not required by the standard), then if Old_Handler is not **null**, it may be called to execute the previous handler. This provides a way to cascade application interrupt handlers. However, the default handler cannot be cascaded this way (Old_Handler must be **null** for the default handler). 18.a.1/1

The Detach_Handler procedure restores the default treatment for the specified interrupt. 19

For all operations defined in this package that take a parameter of type Interrupt_ID, with the exception of Is_Reserved and Reference, a check is made that the specified interrupt is not reserved. {*Program_Error (raised by failure of run-time check)*} Program_Error is raised if this check fails. 20

If, by using the Attach_Handler, Detach_Handler, or Exchange_Handler procedures, an attempt is made to detach a handler that was attached statically (using the pragma Attach_Handler), the handler is not detached and Program_Error is raised. {*Program_Error (raised by failure of run-time check)*} 21

{*AI95-00434-01*} The Reference function returns a value of type System.Address that can be used to attach a task entry via an address clause (see J.7.1) to the interrupt specified by Interrupt. This function raises Program_Error if attaching task entries to interrupts (or to this particular interrupt) is not supported. {*Program_Error (raised by failure of run-time check)*} 22/2

Implementation Requirements

At no time during attachment or exchange of handlers shall the current handler of the corresponding interrupt be undefined. 23

Documentation Requirements

{*AI95-00434-01*} If the Ceiling_Locking policy (see D.3) is in effect, the implementation shall document the default ceiling priority assigned to a protected object that contains either the Attach_Handler or Interrupt_Handler pragmas, but not the Interrupt_Priority pragma. [This default need not be the same for all interrupts.] 24/2

Documentation Requirement: If the Ceiling_Locking policy is in effect, the default ceiling priority for a protected object that contains an interrupt handler pragma. 24.a/2

Implementation Advice

If implementation-defined forms of interrupt handler procedures are supported, such as protected procedures with parameters, then for each such form of a handler, a type analogous to Parameterless_-Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts. 25

Implementation Advice: If implementation-defined forms of interrupt handler procedures are supported, then for each such form of a handler, a type analogous to Parameterless_Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts. 25.a/2

NOTES

26 8 The package Interrupts.Names contains implementation-defined names (and constant values) for the interrupts that are supported by the implementation.

Examples

27 *Example of interrupt handlers:*

28
```
Device_Priority : constant
  array (1..5) of System.Interrupt_Priority := ( ... );
protected type Device_Interface
  (Int_ID : Ada.Interrupts.Interrupt_ID) is
  procedure Handler;
  pragma Attach_Handler(Handler, Int_ID);
  ...
  pragma Interrupt_Priority(Device_Priority(Int_ID));
end Device_Interface;
  ...
Device_1_Driver : Device_Interface(1);
  ...
Device_5_Driver : Device_Interface(5);
  ...
```

Wording Changes from Ada 95

28.a/2 {*8652/0069*} {*AI95-00166-01*} **Corrigendum:** Clarified that the value returned by Current_Handler and Exchange_Handler for the default treatment is null.

C.4 Preelaboration Requirements

1 [This clause specifies additional implementation and documentation requirements for the Preelaborate pragma (see 10.2.1).]

Implementation Requirements

2 The implementation shall not incur any run-time overhead for the elaboration checks of subprograms and protected_bodies declared in preelaborated library units.

3 The implementation shall not execute any memory write operations after load time for the elaboration of constant objects declared immediately within the declarative region of a preelaborated library package, so long as the subtype and initial expression (or default initial expressions if initialized by default) of the object_declaration satisfy the following restrictions. {*load time*} The meaning of *load time* is implementation defined.

3.a **Discussion:** On systems where the image of the partition is initially copied from disk to RAM, or from ROM to RAM, prior to starting execution of the partition, the intention is that “load time” consist of this initial copying step. On other systems, load time and run time might actually be interspersed.

4 • Any subtype_mark denotes a statically constrained subtype, with statically constrained subcomponents, if any;

4.1/2 • {*AI95-00161-01*} no subtype_mark denotes a controlled type, a private type, a private extension, a generic formal private type, a generic formal derived type, or a descendant of such a type;

4.a.1/2 **Reason:** For an implementation that uses the registration method of finalization, a controlled object will require some code executed to register the object at the appropriate point. The other types are those that *might* have a controlled component. None of these types were allowed in preelaborated units in Ada 95. These types are covered by the Implementation Advice, of course, so they should still execute as little code as possible.

5 • any constraint is a static constraint;

6 • any allocator is for an access-to-constant type;

7 • any uses of predefined operators appear only within static expressions;

- any primaries that are names, other than attribute_references for the Access or Address attributes, appear only within static expressions; 8

  Ramification: This cuts out attribute_references that are not static, except for Access and Address. 8.a

- any name that is not part of a static expression is an expanded name or direct_name that statically denotes some entity; 9

  Ramification: This cuts out function_calls and type_conversions that are not static, including calls on attribute functions like 'Image and 'Value. 9.a

- any discrete_choice of an array_aggregate is static; 10
- no language-defined check associated with the elaboration of the object_declaration can fail. 11

  Reason: {*AI95-00114-01*} The intent is that aggregates all of whose scalar subcomponents are static and all of whose access subcomponents are **null**, allocators for access-to-constant types, or X'Access, will be supported with no run-time code generated. 11.a/2

Documentation Requirements

The implementation shall document any circumstances under which the elaboration of a preelaborated package causes code to be executed at run time. 12

Documentation Requirement: Any circumstances when the elaboration of a preelaborated package causes code to be executed. 12.a/2

The implementation shall document whether the method used for initialization of preelaborated variables allows a partition to be restarted without reloading. 13

Documentation Requirement: Whether a partition can be restarted without reloading. 13.a.1/2

This paragraph was deleted. 13.a/2

Discussion: {*AI95-00114-01*} This covers the issue of the run-time system itself being restartable, so that need not be a separate Documentation Requirement. 13.b/2

Implementation Advice

It is recommended that preelaborated packages be implemented in such a way that there should be little or no code executed at run time for the elaboration of entities not already covered by the Implementation Requirements. 14

Implementation Advice: Preelaborated packages should be implemented such that little or no code is executed at run time for the elaboration of entities. 14.a/2

Wording Changes from Ada 95

{*AI95-00161-01*} Added wording to exclude the additional kinds of types allowed in preelaborated units from the Implementation Requirements. 14.b/2

C.5 Pragma Discard_Names

[A pragma Discard_Names may be used to request a reduction in storage used for the names of certain entities.] 1

Syntax

The form of a pragma Discard_Names is as follows: 2

pragma Discard_Names[([On =>] local_name)]; 3

A pragma Discard_Names is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma. {*configuration pragma (Discard_Names)* [partial]} {*pragma, configuration (Discard_Names)* [partial]} 4

Legality Rules

5 The local_name (if present) shall denote a non-derived enumeration [first] subtype, a tagged [first] subtype, or an exception. The pragma applies to the type or exception. Without a local_name, the pragma applies to all such entities declared after the pragma, within the same declarative region. Alternatively, the pragma can be used as a configuration pragma. If the pragma applies to a type, then it applies also to all descendants of the type.

Static Semantics

6 {*representation pragma (Discard_Names)* [partial]} {*pragma, representation (Discard_Names)* [partial]} If a local_name is given, then a pragma Discard_Names is a representation pragma.

7/2 {*AI95-00285-01*} {*AI95-00400-01*} If the pragma applies to an enumeration type, then the semantics of the Wide_Wide_Image and Wide_Wide_Value attributes are implementation defined for that type[; the semantics of Image, Wide_Image, Value, and Wide_Value are still defined in terms of Wide_Wide_Image and Wide_Wide_Value]. In addition, the semantics of Text_IO.Enumeration_IO are implementation defined. If the pragma applies to a tagged type, then the semantics of the Tags.-Wide_Wide_Expanded_Name function are implementation defined for that type[; the semantics of Tags.-Expanded_Name and Tags.Wide_Expanded_Name are still defined in terms of Tags.Wide_Wide_-Expanded_Name]. If the pragma applies to an exception, then the semantics of the Exceptions.-Wide_Wide_Exception_Name function are implementation defined for that exception[; the semantics of Exceptions.Exception_Name and Exceptions.Wide_Exception_Name are still defined in terms of Exceptions.Wide_Wide_Exception_Name].

7.a **Implementation defined:** The semantics of pragma Discard_Names.

7.b **Ramification:** The Width attribute is still defined in terms of Image.

7.c/2 {*AI95-00285-01*} The semantics of S'Wide_Wide_Image and S'Wide_Wide_Value are implementation defined for any subtype of an enumeration type to which the pragma applies. (The pragma actually names the first subtype, of course.)

Implementation Advice

8 If the pragma applies to an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity.

8.a/2 **Implementation Advice:** If pragma Discard_Names applies to an entity, then the amount of storage used for storing names associated with that entity should be reduced.

8.b **Reason:** A typical implementation of the Image attribute for enumeration types is to store a table containing the names of all the enumeration literals. Pragma Discard_Names allows the implementation to avoid storing such a table without having to prove that the Image attribute is never used (which can be difficult in the presence of separate compilation).

8.c We did not specify the semantics of the Image attribute in the presence of this pragma because different semantics might be desirable in different situations. In some cases, it might make sense to use the Image attribute to print out a useful value that can be used to identify the entity given information in compiler-generated listings. In other cases, it might make sense to get an error at compile time or at run time. In cases where memory is plentiful, the simplest implementation makes sense: ignore the pragma. Implementations that are capable of avoiding the extra storage in cases where the Image attribute is never used might also wish to ignore the pragma.

8.d The same applies to the Tags.Expanded_Name and Exceptions.Exception_Name functions.

Wording Changes from Ada 95

8.e/2 {*AI95-00285-01*} {*AI95-00400-01*} Updated the wording to reflect that the double wide image and value functions are now the master versions that the others are defined from.

C.6 Shared Variable Control

1 [This clause specifies representation pragmas that control the use of shared variables.]

Syntax

The form for pragmas Atomic, Volatile, Atomic_Components, and Volatile_Components is as follows: 2

pragma Atomic(local_name); 3

pragma Volatile(local_name); 4

pragma Atomic_Components(*array*_local_name); 5

pragma Volatile_Components(*array*_local_name); 6

{*AI95-00272-01*} {*atomic*} An *atomic* type is one to which a pragma Atomic applies. An *atomic* object (including a component) is one to which a pragma Atomic applies, or a component of an array to which a pragma Atomic_Components applies, or any object of an atomic type, other than objects obtained by evaluating a slice. 7/2

Ramification: {*AI95-00272-01*} A slice of an atomic array object is not itself atomic. That's necessary as executing a read or write of a dynamic number of components in a single instruction is not possible on many targets. 7.a/2

{*volatile*} A *volatile* type is one to which a pragma Volatile applies. A *volatile* object (including a component) is one to which a pragma Volatile applies, or a component of an array to which a pragma Volatile_Components applies, or any object of a volatile type. In addition, every atomic type or object is also defined to be volatile. Finally, if an object is volatile, then so are all of its subcomponents [(the same does not apply to atomic)]. 8

Name Resolution Rules

The local_name in an Atomic or Volatile pragma shall resolve to denote either an object_declaration, a non-inherited component_declaration, or a full_type_declaration. The *array*_local_name in an Atomic_Components or Volatile_Components pragma shall resolve to denote the declaration of an array type or an array object of an anonymous type. 9

Legality Rules

{*indivisible*} It is illegal to apply either an Atomic or Atomic_Components pragma to an object or type if the implementation cannot support the indivisible reads and updates required by the pragma (see below). 10

It is illegal to specify the Size attribute of an atomic object, the Component_Size attribute for an array type with atomic components, or the layout attributes of an atomic component, in a way that prevents the implementation from performing the required indivisible reads and updates. 11

If an atomic object is passed as a parameter, then the type of the formal parameter shall either be atomic or allow pass by copy [(that is, not be a nonatomic by-reference type)]. If an atomic object is used as an actual for a generic formal object of mode **in out**, then the type of the generic formal object shall be atomic. If the prefix of an attribute_reference for an Access attribute denotes an atomic object [(including a component)], then the designated type of the resulting access type shall be atomic. If an atomic type is used as an actual for a generic formal derived type, then the ancestor of the formal type shall be atomic or allow pass by copy. Corresponding rules apply to volatile objects and types. 12

If a pragma Volatile, Volatile_Components, Atomic, or Atomic_Components applies to a stand-alone constant object, then a pragma Import shall also apply to it. 13

Ramification: Hence, no initialization expression is allowed for such a constant. Note that a constant that is atomic or volatile because of its type is allowed. 13.a

Reason: Stand-alone constants that are explicitly specified as Atomic or Volatile only make sense if they are being manipulated outside the Ada program. From the Ada perspective the object is read-only. Nevertheless, if imported and atomic or volatile, the implementation should presume it might be altered externally. For an imported stand-alone constant that is not atomic or volatile, the implementation can assume that it will not be altered. 13.b

Static Semantics

14 {*representation pragma (Atomic)* [partial]} {*pragma, representation (Atomic)* [partial]} {*representation pragma (Volatile)* [partial]} {*pragma, representation (Volatile)* [partial]} {*representation pragma (Atomic_Components)* [partial]} {*pragma, representation (Atomic_Components)* [partial]} {*representation pragma (Volatile_Components)* [partial]} {*pragma, representation (Volatile_Components)* [partial]} These pragmas are representation pragmas (see 13.1).

Dynamic Semantics

15 For an atomic object (including an atomic component) all reads and updates of the object as a whole are indivisible.

16 For a volatile object all reads and updates of the object as a whole are performed directly to memory.

16.a **Implementation Note:** This precludes any use of register temporaries, caches, and other similar optimizations for that object.

17 {*sequential (actions)*} Two actions are sequential (see 9.10) if each is the read or update of the same atomic object.

18 {*by-reference type (atomic or volatile)* [partial]} If a type is atomic or volatile and it is not a by-copy type, then the type is defined to be a by-reference type. If any subcomponent of a type is atomic or volatile, then the type is defined to be a by-reference type.

19 If an actual parameter is atomic or volatile, and the corresponding formal parameter is not, then the parameter is passed by copy.

19.a **Implementation Note:** Note that in the case where such a parameter is normally passed by reference, a copy of the actual will have to be produced at the call-site, and a pointer to the copy passed to the formal parameter. If the actual is atomic, any copying has to use indivisible read on the way in, and indivisible write on the way out.

19.b **Reason:** It has to be known at compile time whether an atomic or a volatile parameter is to be passed by copy or by reference. For some types, it is unspecified whether parameters are passed by copy or by reference. The above rules further specify the parameter passing rules involving atomic and volatile types and objects.

Implementation Requirements

20 {*external effect (volatile/atomic objects)* [partial]} The external effect of a program (see 1.1.3) is defined to include each read and update of a volatile or atomic object. The implementation shall not generate any memory reads or updates of atomic or volatile objects other than those specified by the program.

20.a **Discussion:** The presumption is that volatile or atomic objects might reside in an “active” part of the address space where each read has a potential side-effect, and at the very least might deliver a different value.

20.b The rule above and the definition of external effect are intended to prevent (at least) the following incorrect optimizations, where V is a volatile variable:

20.c • X:= V; Y:=V; cannot be allowed to be translated as Y:=V; X:=V;

20.d • Deleting redundant loads: X:= V; X:= V; shall read the value of V from memory twice.

20.e • Deleting redundant stores: V:= X; V:= X; shall write into V twice.

20.f • Extra stores: V:= X+Y; should not translate to something like V:= X; V:= V+Y;

20.g • Extra loads: X:= V; Y:= X+Z; X:=X+B; should not translate to something like Y:= V+Z; X:= V+B;

20.h • Reordering of loads from volatile variables: X:= V1; Y:= V2; (whether or not V1 = V2) should not translate to Y:= V2; X:= V1;

20.i • Reordering of stores to volatile variables: V1:= X; V2:= X; should not translate to V2:=X; V1:= X;

21 If a pragma Pack applies to a type any of whose subcomponents are atomic, the implementation shall not pack the atomic subcomponents more tightly than that for which it can support indivisible reads and updates.

21.a **Implementation Note:** A warning might be appropriate if no packing whatsoever can be achieved.

Implementation Advice

22/2 {*AI95-00259-01*} A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others.

22.a/2 **Implementation Advice:** A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others.

22.b/2 **Reason:** Since any object can be a volatile object, including packed array components and bit-mapped record components, we require the above only when it is reasonable to assume that the machine can avoid accessing bits outside of the object.

22.c/2 **Ramification:** This implies that the load or store of a volatile object that meets the above requirement should not be combined with that of any other object, nor should it access any bits not belonging to any other object. This means that the suitability of the implementation for memory-mapped I/O can be determined from its documentation, as any cases where the implementation does not follow Implementation Advice must be documented.

23/2 {*AI95-00259-01*} A load or store of an atomic object should, where possible, be implemented by a single load or store instruction.

23.a/2 **Implementation Advice:** A load or store of an atomic object should be implemented by a single load or store instruction.

NOTES

24 9 An imported volatile or atomic constant behaves as a constant (i.e. read-only) with respect to other parts of the Ada program, but can still be modified by an “external source.”

Incompatibilities With Ada 83

24.a {*incompatibilities with Ada 83*} Pragma Atomic replaces Ada 83's pragma Shared. The name “Shared” was confusing, because the pragma was not used to mark variables as shared.

Wording Changes from Ada 95

24.b/2 {*AI95-00259-01*} Added Implementation Advice to clarify the meaning of Atomic and Volatile in machine terms. The documentation that this advice applies will make the use of Ada implementations more predictable for low-level (such as device register) programming.

24.c/2 {*AI95-00272-01*} Added wording to clarify that a slice of an object of an atomic type is not atomic, just like a component of an atomic type is not (necessarily) atomic.

C.7 Task Information

1/2 {*AI95-00266-02*} [This clause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined. Finally, a package that associates termination procedures with a task or set of tasks is defined.]

Wording Changes from Ada 95

1.a/2 {*AI95-00266-02*} The title and text here were updated to reflect the addition of task termination procedures to this clause.

C.7.1 The Package Task_Identification

Static Semantics

1 The following language-defined library package exists:

2/2
```
{AI95-00362-01} package Ada.Task_Identification is
   pragma Preelaborate(Task_Identification);
   type Task_Id is private;
   pragma Preelaborable_Initialization (Task_Id);
   Null_Task_Id : constant Task_Id;
   function  "=" (Left, Right : Task_Id) return Boolean;
```

3/1
```
{8652/0070} {AI95-00101-01}    function  Image        (T : Task_Id) return String;
   function  Current_Task return Task_Id;
   procedure Abort_Task   (T : in Task_Id);
```

4
```
   function  Is_Terminated(T : Task_Id) return Boolean;
   function  Is_Callable  (T : Task_Id) return Boolean;
private
   ... -- not specified by the language
end Ada.Task_Identification;
```

Dynamic Semantics

5 A value of the type Task_Id identifies an existent task. The constant Null_Task_Id does not identify any task. Each object of the type Task_Id is default initialized to the value of Null_Task_Id.

6 The function "=" returns True if and only if Left and Right identify the same task or both have the value Null_Task_Id.

7 The function Image returns an implementation-defined string that identifies T. If T equals Null_Task_Id, Image returns an empty string.

7.a **Implementation defined:** The result of the Task_Identification.Image attribute.

8 The function Current_Task returns a value that identifies the calling task.

9 The effect of Abort_Task is the same as the abort_statement for the task identified by T. [In addition, if T identifies the environment task, the entire partition is aborted, See E.1.]

10 The functions Is_Terminated and Is_Callable return the value of the corresponding attribute of the task identified by T.

10.a.1/1 **Ramification:** {*8652/0115*} {*AI95-00206-01*} These routines can be called with an argument identifying the environment task. Is_Terminated will always be False for such a call, but Is_Callable (usually True) could be False if the environment task is waiting for the termination of dependent tasks. Thus, a dependent task can use Is_Callable to determine if the main subprogram has completed.

11 For a prefix T that is of a task type [(after any implicit dereference)], the following attribute is defined:

12 T'Identity Yields a value of the type Task_Id that identifies the task denoted by T.

13 For a prefix E that denotes an entry_declaration, the following attribute is defined:

14 E'Caller Yields a value of the type Task_Id that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an entry_body or accept_statement corresponding to the entry_declaration denoted by E.

15 {*Program_Error (raised by failure of run-time check)*} Program_Error is raised if a value of Null_Task_Id is passed as a parameter to Abort_Task, Is_Terminated, and Is_Callable.

16 {*potentially blocking operation (Abort_Task)* [partial]} {*blocking, potentially (Abort_Task)* [partial]} Abort_Task is a potentially blocking operation (see 9.5.1).

Bounded (Run-Time) Errors

17/2 {*AI95-00237-01*} {*bounded error (cause)* [partial]} It is a bounded error to call the Current_Task function from an entry body, interrupt handler, or finalization of a task attribute. {*Program_Error (raised by failure of run-time check)*} Program_Error is raised, or an implementation-defined value of the type Task_Id is returned.

17.a/2 **Implementation defined:** The value of Current_Task when in a protected entry, interrupt handler, or finalization of a task attribute.

17.b **Implementation Note:** This value could be Null_Task_Id, or the ID of some user task, or that of an internal task created by the implementation.

Ramification: {*AI95-00237-01*} An entry barrier is syntactically part of an entry_body, so a call to Current_Task from an entry barrier is also covered by this rule. 17.c/2

Erroneous Execution

{*erroneous execution (cause)* [partial]} If a value of Task_Id is passed as a parameter to any of the operations declared in this package (or any language-defined child of this package), and the corresponding task object no longer exists, the execution of the program is erroneous. 18

Documentation Requirements

The implementation shall document the effect of calling Current_Task from an entry body or interrupt handler. 19

This paragraph was deleted. 19.a/2

Documentation Requirement: The effect of calling Current_Task from an entry body or interrupt handler. 19.b/2

NOTES

10 This package is intended for use in writing user-defined task scheduling packages and constructing server tasks. Current_Task can be used in conjunction with other operations requiring a task as an argument such as Set_Priority (see D.5). 20

11 The function Current_Task and the attribute Caller can return a Task_Id value that identifies the environment task. 21

Extensions to Ada 95

{*AI95-00362-01*} Task_Identification is now preelaborated, so it can be used in preelaborated units. 21.a/2

Wording Changes from Ada 95

{*8652/0070*} {*AI95-00101-01*} **Corrigendum:** Corrected the mode of the parameter to Abort_Task to **in**. 21.b/2

{*AI95-00237-01*} Corrected the wording to include finalization of a task attribute in the bounded error case; we don't want to specify which task does these operations. 21.c/2

C.7.2 The Package Task_Attributes

Static Semantics

The following language-defined generic library package exists: 1

```
with Ada.Task_Identification; use Ada.Task_Identification;        2
generic
   type Attribute is private;
   Initial_Value : in Attribute;
package Ada.Task_Attributes is

   type Attribute_Handle is access all Attribute;                   3

   function Value(T : Task_Id := Current_Task)                       4
     return Attribute;

   function Reference(T : Task_Id := Current_Task)                   5
     return Attribute_Handle;

   procedure Set_Value(Val : in Attribute;                           6
                       T : in Task_Id := Current_Task);
   procedure Reinitialize(T : in Task_Id := Current_Task);

end Ada.Task_Attributes;                                             7
```

Dynamic Semantics

When an instance of Task_Attributes is elaborated in a given active partition, an object of the actual type corresponding to the formal type Attribute is implicitly created for each task (of that partition) that exists and is not yet terminated. This object acts as a user-defined attribute of the task. A task created previously in the partition and not yet terminated has this attribute from that point on. Each task subsequently created 8 in the partition will have this attribute when created. In all these cases, the initial value of the given attribute is Initial_Value.

9 The Value operation returns the value of the corresponding attribute of T.

10 The Reference operation returns an access value that designates the corresponding attribute of T.

11 The Set_Value operation performs any finalization on the old value of the attribute of T and assigns Val to that attribute (see 5.2 and 7.6).

12 The effect of the Reinitialize operation is the same as Set_Value where the Val parameter is replaced with Initial_Value.

12.a **Implementation Note:** In most cases, the attribute memory can be reclaimed at this point.

13 {*Tasking_Error (raised by failure of run-time check)*} For all the operations declared in this package, Tasking_Error is raised if the task identified by T is terminated. {*Program_Error (raised by failure of run-time check)*} Program_Error is raised if the value of T is Null_Task_Id.

13.1/2 {*AI95-00237-01*} After a task has terminated, all of its attributes are finalized, unless they have been finalized earlier. When the master of an instantiation of Ada.Task_Attributes is finalized, the corresponding attribute of each task is finalized, unless it has been finalized earlier.

13.a/2 **Reason:** This is necessary so that a task attribute does not outlive its type. For instance, that's possible if the instantiation is nested, and the attribute is on a library-level task.

13.b/2 **Ramification:** The task owning an attribute cannot, in general, finalize that attribute. That's because the attributes are finalized *after* the task is terminated; moreover, a task may have attributes as soon as it is created; the task may never even have been activated.

Bounded (Run-Time) Errors

13.2/1 {*8652/0071*} {*AI95-00165-01*} {*bounded error (cause)* [partial]} If the package Ada.Task_Attributes is instantiated with a controlled type and the controlled type has user-defined Adjust or Finalize operations that in turn access task attributes by any of the above operations, then a call of Set_Value of the instantiated package constitutes a bounded error. The call may perform as expected or may result in forever blocking the calling task and subsequently some or all tasks of the partition.

Erroneous Execution

14 {*erroneous execution (cause)* [partial]} It is erroneous to dereference the access value returned by a given call on Reference after a subsequent call on Reinitialize for the same task attribute, or after the associated task terminates.

14.a **Reason:** This allows the storage to be reclaimed for the object associated with an attribute upon Reinitialize or task termination.

15 {*erroneous execution (cause)* [partial]} If a value of Task_Id is passed as a parameter to any of the operations declared in this package and the corresponding task object no longer exists, the execution of the program is erroneous.

15.1/2 {*8652/0071*} {*AI95-00165-01*} {*AI95-00237-01*} {*erroneous execution (cause)* [partial]} An access to a task attribute via a value of type Attribute_Handle is erroneous if executed concurrently with another such access or a call of any of the operations declared in package Task_Attributes. An access to a task attribute is erroneous if executed concurrently with or after the finalization of the task attribute.

15.a.1/1 **Reason:** There is no requirement of atomicity on accesses via a value of type Attribute_Handle.

15.a.2/2 **Ramification:** A task attribute can only be accessed after finalization through a value of type Attribute_Handle. Operations in package Task_Attributes cannot be used to access a task attribute after finalization, because either the master of the instance has been or is in the process of being left (in which case the instance is out of scope and thus cannot be called), or the associated task is already terminated (in which case Tasking_Error is raised for any attempt to call a task attribute operation).

Implementation Requirements

{*8652/0071*} {*AI95-00165-01*} For a given attribute of a given task, the implementation shall perform the operations declared in this package atomically with respect to any of these operations of the same attribute of the same task. The granularity of any locking mechanism necessary to achieve such atomicity is implementation defined. 16/1

Implementation defined: Granularity of locking for Task_Attributes. 16.a.1/1

Ramification: Hence, other than by dereferencing an access value returned by Reference, an attribute of a given task can be safely read and updated concurrently by multiple tasks. 16.a

{*AI95-00237-01*} After task attributes are finalized, the implementation shall reclaim any storage associated with the attributes. 17/2

Documentation Requirements

The implementation shall document the limit on the number of attributes per task, if any, and the limit on the total storage for attribute values per task, if such a limit exists. 18

In addition, if these limits can be configured, the implementation shall document how to configure them. 19

This paragraph was deleted. 19.a/2

Documentation Requirement: For package Task_Attributes, limits on the number and size of task attributes, and how to configure any limits. 19.b/2

Metrics

{*AI95-00434-01*} The implementation shall document the following metrics: A task calling the following subprograms shall execute at a sufficiently high priority as to not be preempted during the measurement period. This period shall start just before issuing the call and end just after the call completes. If the attributes of task T are accessed by the measurement tests, no other task shall access attributes of that task during the measurement period. For all measurements described here, the Attribute type shall be a scalar type whose size is equal to the size of the predefined type Integer. For each measurement, two cases shall be documented: one where the accessed attributes are of the calling task [(that is, the default value for the T parameter is used)], and the other, where T identifies another, non-terminated, task. 20/2

The following calls (to subprograms in the Task_Attributes package) shall be measured: 21

- a call to Value, where the return value is Initial_Value; 22
- a call to Value, where the return value is not equal to Initial_Value; 23
- a call to Reference, where the return value designates a value equal to Initial_Value; 24
- a call to Reference, where the return value designates a value not equal to Initial_Value; 25
- {*AI95-00434-01*} a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is equal to Initial_Value; 26/2
- a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is not equal to Initial_Value. 27

Documentation Requirement: The metrics for the Task_Attributes package. 27.a/2

Implementation Permissions

An implementation need not actually create the object corresponding to a task attribute until its value is set to something other than that of Initial_Value, or until Reference is called for the task attribute. Similarly, when the value of the attribute is to be reinitialized to that of Initial_Value, the object may 28 instead be finalized and its storage reclaimed, to be recreated when needed later. While the object does not exist, the function Value may simply return Initial_Value, rather than implicitly creating the object.

28.a **Discussion:** The effect of this permission can only be observed if the assignment operation for the corresponding type has side-effects.

28.b/2 **Implementation Note:** {*AI95-00114-01*} This permission means that even though every task has every attribute, storage need only be allocated for those attributes for which function Reference has been invoked or set to a value other than that of Initial_Value.

29 An implementation is allowed to place restrictions on the maximum number of attributes a task may have, the maximum size of each attribute, and the total storage size allocated for all the attributes of a task.

Implementation Advice

30/2 {*AI95-00434-01*} Some implementations are targeted to domains in which memory use at run time must be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the attributes of a task, or by using the pre-allocated storage for the first N attribute objects, and the heap for the others. In the latter case, N should be documented.

30.a/2 **Implementation Advice:** If the target domain requires deterministic memory use at run time, storage for task attributes should be pre-allocated statically and the number of attributes pre-allocated should be documented.

30.b/2 **Discussion:** We don't mention “restrictions on the size and number” (that is, limits) in the text for the Annex, because it is covered by the Documentation Requirement above, and we try not to repeat requirements in the Annex (they're enough work to meet without having to do things twice).

30.1/2 {*AI95-00237-01*} Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination.

30.c/2 **Implementation Advice:** Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination.

30.d/2 **Reason:** {*AI95-00237-01*} This is necessary because the normative wording only says that attributes are finalized “after” task termination. Without this advice, waiting until the instance is finalized would meet the requirements (it is after termination, but may be a very long time after termination). We can't say anything more specific than this, as we do not want to require the overhead of an interaction with the tasking system to be done at a specific point.

NOTES

31 12 An attribute always exists (after instantiation), and has the initial value. It need not occupy memory until the first operation that potentially changes the attribute value. The same holds true after Reinitialize.

32 13 The result of the Reference function should be used with care; it is always safe to use that result in the task body whose attribute is being accessed. However, when the result is being used by another task, the programmer must make sure that the task whose attribute is being accessed is not yet terminated. Failing to do so could make the program execution erroneous.

33/2 *This paragraph was deleted.*{*AI95-00434-01*}

Wording Changes from Ada 95

33.a/2 {*8652/0071*} {*AI95-00165-01*} **Corrigendum:** Clarified that use of task attribute operations from within a task attribute operation (by an Adjust or Finalize call) is a bounded error, and that concurrent use of attribute handles is erroneous.

33.b/2 {*AI95-00237-01*} Clarified the wording so that the finalization takes place after the termination of the task or when the instance is finalized (whichever is sooner).

C.7.3 The Package Task_Termination

Static Semantics

{*AI95-00266-02*} The following language-defined library package exists: 1/2

2/2
```
with Ada.Task_Identification;
with Ada.Exceptions;
package Ada.Task_Termination is
   pragma Preelaborate(Task_Termination);
```

3/2
```
   type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);
```

4/2
```
   type Termination_Handler is access protected procedure
     (Cause : in Cause_Of_Termination;
      T     : in Ada.Task_Identification.Task_Id;
      X     : in Ada.Exceptions.Exception_Occurrence);
```

5/2
```
   procedure Set_Dependents_Fallback_Handler
     (Handler: in Termination_Handler);
   function Current_Task_Fallback_Handler return Termination_Handler;
```

6/2
```
   procedure Set_Specific_Handler
     (T       : in Ada.Task_Identification.Task_Id;
      Handler : in Termination_Handler);
   function Specific_Handler (T : Ada.Task_Identification.Task_Id)
      return Termination_Handler;
```

7/2
```
end Ada.Task_Termination;
```

Dynamic Semantics

{*AI95-00266-02*} {*termination handler*} {*handler (termination)*} The type Termination_Handler identifies a protected procedure to be executed by the implementation when a task terminates. Such a protected procedure is called a *handler*. In all cases T identifies the task that is terminating. If the task terminates due to completing the last statement of its body, or as a result of waiting on a terminate alternative, then Cause is set to Normal and X is set to Null_Occurrence. If the task terminates because it is being aborted, then Cause is set to Abnormal and X is set to Null_Occurrence. If the task terminates because of an exception raised by the execution of its task_body, then Cause is set to Unhandled_Exception and X is set to the associated exception occurrence. 8/2

{*AI95-00266-02*} {*fall-back handler*} {*termination handler (fall-back)*} {*specific handler*} {*termination handler (specific)*} {*set (termination handler)*} {*cleared (termination handler)*} Each task has two termination handlers, a *fall-back handler* and a *specific handler*. The specific handler applies only to the task itself, while the fall-back handler applies only to the dependent tasks of the task. A handler is said to be *set* if it is associated with a non-null value of type Termination_Handler, and *cleared* otherwise. When a task is created, its specific handler and fall-back handler are cleared. 9/2

{*AI95-00266-02*} The procedure Set_Dependents_Fallback_Handler changes the fall-back handler for the calling task; if Handler is **null**, that fall-back handler is cleared, otherwise it is set to be Handler.**all**. If a fall-back handler had previously been set it is replaced. 10/2

{*AI95-00266-02*} The function Current_Task_Fallback_Handler returns the fall-back handler that is currently set for the calling task, if one is set; otherwise it returns **null**. 11/2

{*AI95-00266-02*} The procedure Set_Specific_Handler changes the specific handler for the task identified by T; if Handler is **null**, that specific handler is cleared, otherwise it is set to be Handler.**all**. If a specific handler had previously been set it is replaced. 12/2

{*AI95-00266-02*} The function Specific_Handler returns the specific handler that is currently set for the task identified by T, if one is set; otherwise it returns **null**. 13/2

14/2 {*AI95-00266-02*} As part of the finalization of a task_body, after performing the actions specified in 7.6 for finalization of a master, the specific handler for the task, if one is set, is executed. If the specific handler is cleared, a search for a fall-back handler proceeds by recursively following the master relationship for the task. If a task is found whose fall-back handler is set, that handler is executed; otherwise, no handler is executed.

15/2 {*AI95-00266-02*} For Set_Specific_Handler or Specific_Handler, Tasking_Error is raised if the task identified by T has already terminated. Program_Error is raised if the value of T is Ada.Task_Identification.Null_Task_Id.

16/2 {*AI95-00266-02*} An exception propagated from a handler that is invoked as part of the termination of a task has no effect.

Erroneous Execution

17/2 {*AI95-00266-02*} For a call of Set_Specific_Handler or Specific_Handler, if the task identified by T no longer exists, the execution of the program is erroneous.

Extensions to Ada 95

17.a/2 {*AI95-00266-02*} {*extensions to Ada 95*} Package Task_Termination is new.

Annex D
(normative)
Real-Time Systems

{*real-time systems*} {*embedded systems*} This Annex specifies additional characteristics of Ada implementations intended for real-time systems software. To conform to this Annex, an implementation shall also conform to the Systems Programming Annex. 1

Metrics

The metrics are documentation requirements; an implementation shall document the values of the language-defined metrics for at least one configuration [of hardware or an underlying system] supported by the implementation, and shall document the details of that configuration. 2

This paragraph was deleted. 2.a/2

Documentation Requirement: The details of the configuration used to generate the values of all metrics. 2.a.1/2

Reason: The actual values of the metrics are likely to depend on hardware configuration details that are variable and generally outside the control of a compiler vendor. 2.b

The metrics do not necessarily yield a simple number. [For some, a range is more suitable, for others a formula dependent on some parameter is appropriate, and for others, it may be more suitable to break the metric into several cases.] Unless specified otherwise, the metrics in this annex are expressed in processor clock cycles. For metrics that require documentation of an upper bound, if there is no upper bound, the implementation shall report that the metric is unbounded. 3

Discussion: There are several good reasons to specify metrics in seconds; there are however equally good reasons to specify them in processor clock cycles. In defining the metrics, we have tried to strike a balance on a case-by-case basis. 3.a

It has been suggested that all metrics should be given names, so that “data-sheets” could be formulated and published by vendors. However the paragraph number can serve that purpose. 3.b

NOTES

1 The specification of the metrics makes a distinction between upper bounds and simple execution times. Where something is just specified as “the execution time of” a piece of code, this leaves one the freedom to choose a nonpathological case. This kind of metric is of the form “there exists a program such that the value of the metric is V”. Conversely, the meaning of upper bounds is “there is no program such that the value of the metric is greater than V”. This kind of metric can only be partially tested, by finding the value of V for one or more test programs. 4

2 The metrics do not cover the whole language; they are limited to features that are specified in Annex C, “Systems Programming” and in this Annex. The metrics are intended to provide guidance to potential users as to whether a particular implementation of such a feature is going to be adequate for a particular real-time application. As such, the metrics are aimed at known implementation choices that can result in significant performance differences. 5

3 The purpose of the metrics is not necessarily to provide fine-grained quantitative results or to serve as a comparison between different implementations on the same or different platforms. Instead, their goal is rather qualitative; to define a standard set of approximate values that can be measured and used to estimate the general suitability of an implementation, or to evaluate the comparative utility of certain features of an implementation for a particular real-time application. 6

Extensions to Ada 83

{*extensions to Ada 83*} This Annex is new to Ada 95. 6.a

D.1 Task Priorities

[This clause specifies the priority model for real-time systems. In addition, the methods for specifying priorities are defined.] 1

Syntax

2 The form of a pragma Priority is as follows:

3 **pragma** Priority(expression);

4 The form of a pragma Interrupt_Priority is as follows:

5 **pragma** Interrupt_Priority[(expression)];

Name Resolution Rules

6 {*expected type (Priority pragma argument)* [partial]} {*expected type (Interrupt_Priority pragma argument)* [partial]} The expected type for the expression in a Priority or Interrupt_Priority pragma is Integer.

Legality Rules

7 A Priority pragma is allowed only immediately within a task_definition, a protected_definition, or the declarative_part of a subprogram_body. An Interrupt_Priority pragma is allowed only immediately within a task_definition or a protected_definition. At most one such pragma shall appear within a given construct.

8 For a Priority pragma that appears in the declarative_part of a subprogram_body, the expression shall be static, and its value shall be in the range of System.Priority.

8.a **Reason:** This value is needed before it gets elaborated, when the environment task starts executing.

Static Semantics

9 The following declarations exist in package System:

10
```
subtype Any_Priority is Integer range implementation-defined;
subtype Priority is Any_Priority
   range Any_Priority'First .. implementation-defined;
subtype Interrupt_Priority is Any_Priority
   range Priority'Last+1 .. Any_Priority'Last;
```

11
```
Default_Priority : constant Priority := (Priority'First + Priority'Last)/2;
```

11.a **Implementation defined:** The declarations of Any_Priority and Priority.

12 The full range of priority values supported by an implementation is specified by the subtype Any_Priority. The subrange of priority values that are high enough to require the blocking of one or more interrupts is specified by the subtype Interrupt_Priority. [The subrange of priority values below System.-Interrupt_Priority'First is specified by the subtype System.Priority.]

13 The priority specified by a Priority or Interrupt_Priority pragma is the value of the expression in the pragma, if any. If there is no expression in an Interrupt_Priority pragma, the priority value is Interrupt_Priority'Last.

Dynamic Semantics

14 A Priority pragma has no effect if it occurs in the declarative_part of the subprogram_body of a subprogram other than the main subprogram.

15 {*task priority*} {*priority*} {*priority inheritance*} {*base priority*} {*active priority*} A *task priority* is an integer value that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. Unless otherwise specified, whenever tasks compete for processors or other implementation-defined resources, the resources are allocated to the task with the highest priority value. The *base priority* of a task is the priority with which it was created, or to which it was later set by Dynamic_Priorities.Set_Priority (see D.5). At all times, a task also has an *active priority*, which generally reflects its base priority as well as any priority it inherits from other sources. *Priority inheritance* is the

process by which the priority of a task or other entity (e.g. a protected object; see D.3) is used in the evaluation of another task's active priority.

Implementation defined: Implementation-defined execution resources. 15.a

The effect of specifying such a pragma in a protected_definition is discussed in D.3. 16

{*creation (of a task object)*} The expression in a Priority or Interrupt_Priority pragma that appears in a task_definition is evaluated for each task object (see 9.1). For a Priority pragma, the value of the expression is converted to the subtype Priority; for an Interrupt_Priority pragma, this value is converted to the subtype Any_Priority. The priority value is then associated with the task object whose task_definition contains the pragma. {*implicit subtype conversion (pragma Priority)* [partial]} {*implicit subtype conversion (pragma Interrupt_Priority)* [partial]} 17

Likewise, the priority value is associated with the environment task if the pragma appears in the declarative_part of the main subprogram. 18

The initial value of a task's base priority is specified by default or by means of a Priority or Interrupt_Priority pragma. [After a task is created, its base priority can be changed only by a call to Dynamic_Priorities.Set_Priority (see D.5).] The initial base priority of a task in the absence of a pragma is the base priority of the task that creates it at the time of creation (see 9.1). If a pragma Priority does not apply to the main subprogram, the initial base priority of the environment task is System.Default_Priority. [The task's active priority is used when the task competes for processors. Similarly, the task's active priority is used to determine the task's position in any queue when Priority_Queuing is specified (see D.4).] 19

{*AI95-00357-01*} At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant. For a task that is not held (see D.11), its base priority is a source of priority inheritance unless otherwise specified for a particular task dispatching policy. Other sources of priority inheritance are specified under the following conditions: 20/2

Discussion: Other parts of the annex, e.g. D.11, define other sources of priority inheritance. 20.a

- {*8652/0072*} {*AI95-00092-01*} During activation, a task being activated inherits the active priority that its activator (see 9.2) had at the time the activation was initiated. 21/1
- {*8652/0072*} {*AI95-00092-01*} During rendezvous, the task accepting the entry call inherits the priority of the entry call (see 9.5.3 and D.4). 22/1
- During a protected action on a protected object, a task inherits the ceiling priority of the protected object (see 9.5 and D.3). 23

In all of these cases, the priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists. 24

Implementation Requirements

The range of System.Interrupt_Priority shall include at least one value. 25

The range of System.Priority shall include at least 30 values. 26

NOTES

4 The priority expression can include references to discriminants of the enclosing type. 27

5 It is a consequence of the active priority rules that at the point when a task stops inheriting a priority from another source, its active priority is re-evaluated. This is in addition to other instances described in this Annex for such re-evaluation. 28

6 An implementation may provide a non-standard mode in which tasks inherit priorities under conditions other than those specified above. 29

Ramification: The use of a Priority or Interrupt_Priority pragma does not require the package System to be named in a with_clause for the enclosing compilation_unit. 29.a

Extensions to Ada 83

29.b {*extensions to Ada 83*} The priority of a task is per-object and not per-type.

29.c Priorities need not be static anymore (except for the main subprogram).

Wording Changes from Ada 83

29.d The description of the Priority pragma has been moved to this annex.

Wording Changes from Ada 95

29.e/2 {*8652/0072*} {*AI95-00092-01*} **Corrigendum:** Clarified that dynamic priority changes are not transitive - that is, they don't apply to tasks that are being activated by or in rendezvous with the task that had its priority changed.

29.f/2 {*AI95-00357-01*} Generalized the definition of priority inheritance to take into account the differences between the existing and new dispatching policies.

D.2 Priority Scheduling

1/2 {*AI95-00321-01*} [This clause describes the rules that determine which task is selected for execution when more than one task is ready (see 9).]

Wording Changes from Ada 95

1.a/2 {*AI95-00321-01*} This introduction is simplified in order to reflect the rearrangement and expansion of this clause.

D.2.1 The Task Dispatching Model

1/2 {*AI95-00321-01*} [The task dispatching model specifies task scheduling, based on conceptual priority-ordered ready queues.]

Static Semantics

1.1/2 {*AI95-00355-01*} The following language-defined library package exists:

1.2/2
```
package Ada.Dispatching is
  pragma Pure(Dispatching);
  Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

1.3/2 Dispatching serves as the parent of other language-defined library units concerned with task dispatching.

Dynamic Semantics

2/2 {*AI95-00321-01*} A task can become a *running task* only if it is ready (see 9) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

3 It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.

3.a **Implementation defined:** Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.

4/2 {*AI95-00321-01*} {*task dispatching*} {*dispatching, task*} {*task dispatching point* [distributed]} {*dispatching point* [distributed]} *Task dispatching* is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called *task dispatching points*. A task reaches a task dispatching point whenever it becomes blocked, and when it terminates. [Other task dispatching points are defined throughout this Annex for specific policies.]

4.a **Ramification:** On multiprocessor systems, more than one task can be chosen, at the same time, for execution on more than one processor, as explained below.

{*AI95-00321-01*} {*ready queue*} {*head (of a queue)*} {*tail (of a queue)*} {*ready task*} {*task dispatching policy* [partial]} {*dispatching policy for tasks* [partial]} *Task dispatching policies* are specified in terms of conceptual *ready queues* and task states. A ready queue is an ordered list of ready tasks. The first position in a queue is called the *head of the queue*, and the last position is called the *tail of the queue*. A task is *ready* if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor. 5/2

Discussion: The core language defines a ready task as one that is not blocked. Here we refine this definition and talk about ready queues. 5.a

{*AI95-00321-01*} {*running task*} Each processor also has one *running task*, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs. 6/2

Discussion: There is always at least one task to run, if we count the idle task. 6.a

This paragraph was deleted.{*AI95-00321-01*} 7/2

This paragraph was deleted. 7.a/2

This paragraph was deleted.{*AI95-00321-01*} 8/2

This paragraph was deleted. 8.a/2

Implementation Permissions

{*AI95-00321-01*} An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation-defined effect on task dispatching. 9/2

Implementation defined: The effect of implementation-defined execution resources on task dispatching. 9.a/2

An implementation may place implementation-defined restrictions on tasks whose active priority is in the Interrupt_Priority range. 10

Ramification: For example, on some operating systems, it might be necessary to disallow them altogether. This permission applies to tasks whose priority is set to interrupt level for any reason: via a pragma, via a call to Dynamic_Priorities.Set_Priority, or via priority inheritance. 10.a

{*AI95-00321-01*} [For optimization purposes,] an implementation may alter the points at which task dispatching occurs, in an implementation-defined manner. However, a delay_statement always corresponds to at least one task dispatching point. 10.1/2

NOTES

7 Section 9 specifies under which circumstances a task becomes ready. The ready state is affected by the rules for task activation and termination, delay statements, and entry calls. {*blocked* [partial]} When a task is not ready, it is said to be blocked. 11

8 An example of a possible implementation-defined execution resource is a page of physical memory, which needs to be loaded with a particular page of virtual memory before a task can continue execution. 12

9 The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation. 13

10 While a task is running, it is not on any ready queue. Any time the task that is running on a processor is added to a ready queue, a new running task is selected for that processor. 14

11 In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. [Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.] 15

16 12 The priority of a task is determined by rules specified in this subclause, and under D.1, “Task Priorities”, D.3, “Priority Ceiling Locking”, and D.5, “Dynamic Priorities”.

17/2 13 {*AI95-00321-01*} The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.

Wording Changes from Ada 95

17.a/2 {*AI95-00321-01*} This description is simplified to describe only the parts of the dispatching model common to all policies. In particular, rules about preemption are moved elsewhere. This makes it easier to add other policies (which may not involve preemption).

D.2.2 Task Dispatching Pragmas

0.1/2 {*AI95-00355-01*} [This clause allows a single task dispatching policy to be defined for all priorities, or the range of priorities to be split into subranges that are assigned individual dispatching policies.]

Syntax

1 The form of a pragma Task_Dispatching_Policy is as follows:

2 **pragma** Task_Dispatching_Policy(*policy*_identifier);

2.1/2 {*AI95-00355-01*} The form of a pragma Priority_Specific_Dispatching is as follows:

2.2/2 **pragma** Priority_Specific_Dispatching (
*policy*_identifier, *first_priority*_expression, *last_priority*_expression);

Name Resolution Rules

2.3/2 {*AI95-00355-01*} The expected type for *first_priority*_expression and *last_priority*_expression is Integer.

Legality Rules

3/2 {*AI95-00321-01*} {*AI95-00355-01*} The *policy*_identifier used in a pragma Task_Dispatching_Policy shall be the name of a task dispatching policy.

3.a/2 *This paragraph was deleted.*

3.1/2 {*AI95-00355-01*} The *policy*_identifier used in a pragma Priority_Specific_Dispatching shall be the name of a task dispatching policy.

3.2/2 {*AI95-00355-01*} Both *first_priority*_expression and *last_priority*_expression shall be static expressions in the range of System.Any_Priority; *last_priority*_expression shall have a value greater than or equal to *first_priority*_expression.

Static Semantics

3.3/2 {*AI95-00355-01*} Pragma Task_Dispatching_Policy specifies the single task dispatching policy.

3.4/2 {*AI95-00355-01*} Pragma Priority_Specific_Dispatching specifies the task dispatching policy for the specified range of priorities. Tasks with base priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma have their active priorities determined according to the specified dispatching policy. Tasks with active priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma are dispatched according to the specified dispatching policy.

3.b/2 **Reason:** {*AI95-00355-01*} Each ready queue is managed by exactly one policy. Anything else would be chaos. The ready queue is determined by the active priority. However, how the active priority is calculated is determined by the policy; in order to break out of this circle, we have to say that the active priority is calculated by the method determined by the policy of the base priority.

{*AI95-00355-01*} If a partition contains one or more Priority_Specific_Dispatching pragmas the dispatching policy for priorities not covered by any Priority_Specific_Dispatching pragmas is FIFO_Within_Priorities. 3.5/2

Post-Compilation Rules

{*AI95-00355-01*} {*configuration pragma (Task_Dispatching_Policy)* [partial]} {*pragma, configuration (Task_Dispatching_Policy)* [partial]} A Task_Dispatching_Policy pragma is a configuration pragma. A Priority_Specific_Dispatching pragma is a configuration pragma. {*configuration pragma (Priority_Specific_Dispatching)* [partial]} {*pragma, configuration (Priority_Specific_Dispatching)* [partial]} 4/2

{*AI95-00355-01*} The priority ranges specified in more than one Priority_Specific_Dispatching pragma within the same partition shall not be overlapping. 4.1/2

{*AI95-00355-01*} If a partition contains one or more Priority_Specific_Dispatching pragmas it shall not contain a Task_Dispatching_Policy pragma. 4.2/2

This paragraph was deleted.{*AI95-00333-01*} 5/2

Dynamic Semantics

{*AI95-00355-01*} {*task dispatching policy*} [A *task dispatching policy* specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues.] A single task dispatching policy is specified by a Task_Dispatching_Policy pragma. Pragma Priority_Specific_Dispatching assigns distinct dispatching policies to subranges of System.Any_Priority. 6/2

{*AI95-00355-01*} {*unspecified* [partial]} If neither pragma applies to any of the program units comprising a partition, the task dispatching policy for that partition is unspecified. 6.1/2

{*AI95-00355-01*} If a partition contains one or more Priority_Specific_Dispatching pragmas a task dispatching point occurs for the currently running task of a processor whenever there is a non-empty ready queue for that processor with a higher priority than the priority of the running task. 6.2/2

> **Discussion:** If we have priority specific dispatching then we want preemption across the entire range of priorities. That prevents higher priority tasks from being blocked by lower priority tasks that have a different policy. On the other hand, if we have a single policy for the entire partition, we want the characteristics of that policy to apply for preemption; specifically, we may not require any preemption. Note that policy Non_Preemptive_FIFO_Within_Priorities is not allowed in a priority specific dispatching pragma. 6.a/2

{*AI95-00355-01*} A task that has its base priority changed may move from one dispatching policy to another. It is immediately subject to the new dispatching policy. 6.3/2

> **Ramification:** Once subject to the new dispatching policy, it may be immediately preempted or dispatched, according the rules of the new policy. 6.b/2

Paragraphs 7 through 13 were moved to D.2.3.

Implementation Requirements

{*AI95-00333-01*} {*AI95-00355-01*} An implementation shall allow, for a single partition, both the locking policy (see D.3) to be specified as Ceiling_Locking and also one or more Priority_Specific_Dispatching pragmas to be given. 13.1/2

Documentation Requirements

Paragraphs 14 through 16 were moved to D.2.3.

> *This paragraph was deleted.* 16.a/2

Implementation Permissions

17/2 {*AI95-00256-01*} Implementations are allowed to define other task dispatching policies, but need not support more than one task dispatching policy per partition.

18/2 {*AI95-00355-01*} An implementation need not support pragma Priority_Specific_Dispatching if it is infeasible to support it in the target environment.

18.a/2 **Implementation defined:** Implementation defined task dispatching policies.

NOTES

Paragraphs 19 through 21 were deleted.

Extensions to Ada 95

21.a/2 {*AI95-00333-01*} {*extensions to Ada 95*} **Amendment Correction:** It is no longer required to specify Ceiling_Locking with the language-defined task dispatching policies; we only require that implementations *allow* them to be used together.

21.b/2 {*AI95-00355-01*} **Pragma** Priority_Specific_Dispatching is new; it allows specifying different policies for different priorities.

Wording Changes from Ada 95

21.c/2 {*AI95-00256-01*} Clarified that an implementation need support only one task dispatching policy (of any kind, language-defined or otherwise) per partition.

21.d/2 {*AI95-00321-01*} This description is simplified to describe only the rules for the Task_Dispatching_Policy pragma that are common to all policies. In particular, rules about preemption are moved elsewhere. This makes it easier to add other policies (which may not involve preemption).

D.2.3 Preemptive Dispatching

1/2 {*AI95-00321-01*} [This clause defines a preemptive task dispatching policy.]

Static Semantics

2/2 {*AI95-00355-01*} The *policy*_identifier FIFO_Within_Priorities is a task dispatching policy.

Dynamic Semantics

3/2 {*AI95-00321-01*} When FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

4/2 • {*AI95-00321-01*} When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.

5/2 • When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.

6/2 • When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.

7/2 • When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

7.a/2 **Ramification:** If the delay does result in blocking, the task moves to the “delay queue”, not to the ready queue.

8/2 {*AI95-00321-01*} {*task dispatching point* [partial]} {*dispatching point* [partial]} Each of the events specified above is a task dispatching point (see D.2.1).

{*AI95-00321-01*} A task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. The currently running task is said to be *preempted* and it is added at the head of the ready queue for its active priority.{*preempt (a running task)*} 9/2

Implementation Requirements

{*AI95-00333-01*} An implementation shall allow, for a single partition, both the task dispatching policy to be specified as FIFO_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking. 10/2

Reason: This is the preferred combination of the FIFO_Within_Priorities policy with a locking policy, and we want that combination to be portable. 10.a/2

Documentation Requirements

{*AI95-00321-01*} {*priority inversion*} *Priority inversion* is the duration for which a task remains at the head of the highest priority nonempty ready queue while the processor executes a lower priority task. The implementation shall document: 11/2

- The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and 12/2

Documentation Requirement: The maximum priority inversion a user task can experience from the implementation. 12.a/2

- whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long. 13/2

Documentation Requirement: The amount of time that a task can be preempted for processing on behalf of lower-priority tasks. 13.a/2

NOTES

14 {*AI95-00321-01*} If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task). 14/2

15 {*AI95-00321-01*} Setting the base priority of a ready task causes the task to move to the tail of the queue for its active priority, regardless of whether the active priority of the task actually changes. 15/2

Wording Changes from Ada 95

{*AI95-00321-01*} This subclause is new; it mainly consists of text that was found in D.2.1 and D.2.2 in Ada 95. This was separated out so the definition of additional policies was easier. 15.a/2

{*AI95-00333-01*} We require that implementations allow this policy and Ceiling_Locking together. 15.b/2

{*AI95-00355-01*} We explicitly defined FIFO_Within_Priorities to be a task dispatching policy. 15.c/2

D.2.4 Non-Preemptive Dispatching

{*AI95-00298-01*} [This clause defines a non-preemptive task dispatching policy.] 1/2

Static Semantics

{*AI95-00298-01*} {*AI95-00355-01*} The *policy_*identifier Non_Preemptive_FIFO_Within_Priorities is a task dispatching policy. 2/2

Legality Rules

{*AI95-00355-01*} Non_Preemptive_FIFO_Within_Priorities shall not be specified as the *policy_*identifier of pragma Priority_Specific_Dispatching (see D.2.2). 3/2

Reason: The non-preemptive nature of this policy could cause the policies of higher priority tasks to malfunction, missing deadlines and having unlimited priority inversion. That would render the use of such policies impotent and misleading. As such, this policy only makes sense for a complete system. 3.a/2

Dynamic Semantics

4/2 {*AI95-00298-01*} When Non_Preemptive_FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

5/2
- {*AI95-00298-01*} When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.

6/2
- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority.

7/2
- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.

8/2
- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

8.a/2 **Ramification:** If the delay does result in blocking, the task moves to the “delay queue”, not to the ready queue.

9/2 For this policy, a non-blocking delay_statement is the only non-blocking event that is a task dispatching point (see D.2.1).{*task dispatching point* [partial]} {*dispatching point* [partial]}

Implementation Requirements

10/2 {*AI95-00333-01*} An implementation shall allow, for a single partition, both the task dispatching policy to be specified as Non_Preemptive_FIFO_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

10.a/2 **Reason:** This is the preferred combination of the Non_Preemptive_FIFO_Within_Priorities policy with a locking policy, and we want that combination to be portable.

Implementation Permissions

11/2 {*AI95-00298-01*} Since implementations are allowed to round all ceiling priorities in subrange System.Priority to System.Priority'Last (see D.3), an implementation may allow a task to execute within a protected object without raising its active priority provided the associated protected unit does not contain pragma Interrupt_Priority, Interrupt_Handler, or Attach_Handler.

Extensions to Ada 95

11.a/2 {*AI95-00298-01*} {*AI95-00355-01*} {*extensions to Ada 95*} Policy Non_Preemptive_FIFO_Within_Priorities is new.

D.2.5 Round Robin Dispatching

1/2 {*AI95-00355-01*} [This clause defines the task dispatching policy Round_Robin_Within_Priorities and the package Round_Robin.]

Static Semantics

2/2 {*AI95-00355-01*} The *policy_*identifier Round_Robin_Within_Priorities is a task dispatching policy.

{*AI95-00355-01*} The following language-defined library package exists: 3/2

4/2
```
with System;
with Ada.Real_Time;
package Ada.Dispatching.Round_Robin is
  Default_Quantum : constant Ada.Real_Time.Time_Span :=
              implementation-defined;
  procedure Set_Quantum (Pri     : in System.Priority;
                         Quantum : in Ada.Real_Time.Time_Span);
  procedure Set_Quantum (Low, High : in System.Priority;
                         Quantum   : in Ada.Real_Time.Time_Span);
  function Actual_Quantum (Pri : System.Priority) return Ada.Real_Time.Time_Span;
  function Is_Round_Robin (Pri : System.Priority) return Boolean;
end Ada.Dispatching.Round_Robin;
```

Implementation defined: The value of Default_Quantum in Dispatching.Round_Robin. 4.a.1/2

{*AI95-00355-01*} When task dispatching policy Round_Robin_Within_Priorities is the single policy in effect for a partition, each task with priority in the range of System.Interrupt_Priority is dispatched according to policy FIFO_Within_Priorities. 5/2

Dynamic Semantics

{*AI95-00355-01*} The procedures Set_Quantum set the required Quantum value for a single priority level Pri or a range of priority levels Low .. High. If no quantum is set for a Round Robin priority level, Default_Quantum is used. 6/2

{*AI95-00355-01*} The function Actual_Quantum returns the actual quantum used by the implementation for the priority level Pri. 7/2

{*AI95-00355-01*} The function Is_Round_Robin returns True if priority Pri is covered by task dispatching policy Round_Robin_Within_Priorities; otherwise it returns False. 8/2

{*AI95-00355-01*} A call of Actual_Quantum or Set_Quantum raises exception Dispatching.Dispatching_Policy_Error if a predefined policy other than Round_Robin_Within_Priorities applies to the specified priority or any of the priorities in the specified range. 9/2

{*AI95-00355-01*} For Round_Robin_Within_Priorities, the dispatching rules for FIFO_Within_Priorities apply with the following additional rules: 10/2

- When a task is added or moved to the tail of the ready queue for its base priority, it has an execution time budget equal to the quantum for that priority level. This will also occur when a blocked task becomes executable again. 11/2
- When a task is preempted (by a higher priority task) and is added to the head of the ready queue for its priority level, it retains its remaining budget. 12/2
- While a task is executing, its budget is decreased by the amount of execution time it uses. The accuracy of this accounting is the same as that for execution time clocks (see D.14). 13/2

  Ramification: Note that this happens even when the task is executing at a higher, inherited priority, and even if that higher priority is dispatched by a different policy than round robin. 13.a/2
- When a task has exhausted its budget and is without an inherited priority (and is not executing within a protected operation), it is moved to the tail of the ready queue for its priority level. This is a task dispatching point. 14/2

  Ramification: In this case, it will be given a budget as described in the first bullet. 14.a/2

  The rules for FIFO_Within_Priority (to which these bullets are added) say that a task that has its base priority set to a Round Robin priority is moved to the tail of the ready queue for its new priority level, and then will be given a budget as described in the first bullet. That happens whether or not the task's original base priority was a Round Robin priority. 14.b/2

Implementation Requirements

15/2 {*AI95-00333-01*} {*AI95-00355-01*} An implementation shall allow, for a single partition, both the task dispatching policy to be specified as Round_Robin_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

15.a/2 **Reason:** This is the preferred combination of the Round_Robin_Within_Priorities policy with a locking policy, and we want that combination to be portable.

Documentation Requirements

16/2 {*AI95-00355-01*} An implementation shall document the quantum values supported.

16.a.1/2 **Documentation Requirement:** The quantum values supported for round robin dispatching.

17/2 {*AI95-00355-01*} An implementation shall document the accuracy with which it detects the exhaustion of the budget of a task.

17.a.1/2 **Documentation Requirement:** The accuracy of the detection of the exhaustion of the budget of a task for round robin dispatching.

NOTES

18/2 16 {*AI95-00355-01*} Due to implementation constraints, the quantum value returned by Actual_Quantum might not be identical to that set with Set_Quantum.

19/2 17 {*AI95-00355-01*} A task that executes continuously with an inherited priority will not be subject to round robin dispatching.

Extensions to Ada 95

19.a/2 {*AI95-00355-01*} {*extensions to Ada 95*} Policy Round_Robin_Within_Priorities and package Dispatching.Round_Robin are new.

D.2.6 Earliest Deadline First Dispatching

1/2 {*AI95-00357-01*} The deadline of a task is an indication of the urgency of the task; it represents a point on an ideal physical time line. The deadline might affect how resources are allocated to the task.

2/2 {*AI95-00357-01*} This clause defines a package for representing the deadline of a task and a dispatching policy that defines Earliest Deadline First (EDF) dispatching. A pragma is defined to assign an initial deadline to a task.

2.a/2 **Discussion:** This pragma is the only way of assigning an initial deadline to a task so that its activation can be controlled by EDF scheduling. This is similar to the way pragma Priority is used to give an initial priority to a task.

Language Design Principles

2.b/2 {*AI95-00357-01*} To predict the behavior of a multi-tasking program it is necessary to control access to the processor which is preemptive, and shared objects which are usually non-preemptive and embodied in protected objects. Two common dispatching policies for the processor are fixed priority and EDF. The most effective control over shared objects is via preemption levels. With a pure priority scheme a single notion of priority is used for processor dispatching and preemption levels. With EDF and similar schemes priority is used for preemption levels (only), with another measure used for dispatching. T.P. Baker showed (*Real-Time Systems*, March 1991, vol. 3, num. 1, *Stack-Based Scheduling of Realtime Processes*) that for EDF a newly released task should only preempt the currently running task if it has an earlier deadline and a higher preemption level than any currently “locked” protected object. The rules of this clause implement this scheme including the case where the newly released task should execute before some existing tasks but not preempt the currently executing task.

Syntax

3/2 {*AI95-00357-01*} The form of a pragma Relative_Deadline is as follows:

4/2 **pragma** Relative_Deadline (*relative_deadline*_expression);

Name Resolution Rules

{*AI95-00357-01*} The expected type for *relative_deadline*_expression is Real_Time.Time_Span. 5/2

Legality Rules

{*AI95-00357-01*} A Relative_Deadline pragma is allowed only immediately within a task_definition or the declarative_part of a subprogram_body. At most one such pragma shall appear within a given construct. 6/2

Static Semantics

{*AI95-00357-01*} The *policy*_identifier EDF_Across_Priorities is a task dispatching policy. 7/2

{*AI95-00357-01*} The following language-defined library package exists: 8/2

9/2
```
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF is
  subtype Deadline is Ada.Real_Time.Time;
  Default_Deadline : constant Deadline :=
              Ada.Real_Time.Time_Last;
  procedure Set_Deadline (D : in Deadline;
              T : in Ada.Task_Identification.Task_Id :=
              Ada.Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline (
              Delay_Until_Time : in Ada.Real_Time.Time;
              Deadline_Offset : in Ada.Real_Time.Time_Span);
  function Get_Deadline (T : Ada.Task_Identification.Task_Id :=
              Ada.Task_Identification.Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

Post-Compilation Rules

{*AI95-00357-01*} If the EDF_Across_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition. 10/2

{*AI95-00357-01*} If the EDF_Across_Priorities policy appears in a Priority_Specific_Dispatching pragma (see D.2.2) in a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition. 11/2

Reason: Unlike the other language-defined dispatching policies, the semantic description of EDF_Across_Priorities assumes Ceiling_Locking (and a ceiling priority) in order to make the mapping between deadlines and priorities work. Thus, we require both policies to be specified if EDF is used in the partition. 11.a/2

Dynamic Semantics

{*AI95-00357-01*} A Relative_Deadline pragma has no effect if it occurs in the declarative_part of the subprogram_body of a subprogram other than the main subprogram. 12/2

{*AI95-00357-01*} The initial absolute deadline of a task containing pragma Relative_Deadline is the value of Real_Time.Clock + *relative_deadline*_expression, where the call of Real_Time.Clock is made between task creation and the start of its activation. If there is no Relative_Deadline pragma then the initial absolute deadline of a task is the value of Default_Deadline. [The environment task is also given an initial deadline by this rule.] 13/2

Proof: The environment task is a normal task by 10.2, so of course this rule applies to it. 13.a/2

{*AI95-00357-01*} The procedure Set_Deadline changes the absolute deadline of the task to D. The function Get_Deadline returns the absolute deadline of the task. 14/2

15/2 {*AI95-00357-01*} The procedure Delay_Until_And_Set_Deadline delays the calling task until time Delay_Until_Time. When the task becomes runnable again it will have deadline Delay_Until_Time + Deadline_Offset.

16/2 {*AI95-00357-01*} On a system with a single processor, the setting of the deadline of a task to the new value occurs immediately at the first point that is outside the execution of a protected action. If the task is currently on a ready queue it is removed and re-entered on to the ready queue determined by the rules defined below.

17/2 {*AI95-00357-01*} When EDF_Across_Priorities is specified for priority range *Low*..*High* all ready queues in this range are ordered by deadline. The task at the head of a queue is the one with the earliest deadline.

18/2 {*AI95-00357-01*} A task dispatching point occurs for the currently running task *T* to which policy EDF_Across_Priorities applies:

19/2 • when a change to the deadline of *T* occurs;

20/2 • there is a task on the ready queue for the active priority of *T* with a deadline earlier than the deadline of *T*; or

21/2 • there is a non-empty ready queue for that processor with a higher priority than the active priority of the running task.

22/2 In these cases, the currently running task is said to be preempted and is returned to the ready queue for its active priority.

23/2 {*AI95-00357-01*} For a task *T* to which policy EDF_Across_Priorities applies, the base priority is not a source of priority inheritance; the active priority when first activated or while it is blocked is defined as the maximum of the following:

24/2 • the lowest priority in the range specified as EDF_Across_Priorities that includes the base priority of *T*;

25/2 • the priorities, if any, currently inherited by *T*;

26/2 • the highest priority *P*, if any, less than the base priority of *T* such that one or more tasks are executing within a protected object with ceiling priority *P* and task *T* has an earlier deadline than all such tasks.

26.a/2 **Ramification:** The active priority of *T* might be lower than its base priority.

27/2 {*AI95-00357-01*} When a task *T* is first activated or becomes unblocked, it is added to the ready queue corresponding to this active priority. Until it becomes blocked again, the active priority of *T* remains no less than this value; it will exceed this value only while it is inheriting a higher priority.

27.a/2 **Discussion:** These rules ensure that a task executing in a protected object is preempted only by a task with a shorter deadline and a higher base priority. This matches the traditional preemption level description without the need to define a new kind of protected object locking.

28/2 {*AI95-00357-01*} When the setting of the base priority of a ready task takes effect and the new priority is in a range specified as EDF_Across_Priorities, the task is added to the ready queue corresponding to its new active priority, as determined above.

29/2 {*AI95-00357-01*} For all the operations defined in Dispatching.EDF, Tasking_Error is raised if the task identified by T has terminated. Program_Error is raised if the value of T is Null_Task_Id.

Bounded (Run-Time) Errors

30/2 {*AI95-00357-01*} {*bounded error (cause)* [partial]} If EDF_Across_Priorities is specified for priority range *Low*..*High*, it is a bounded error to declare a protected object with ceiling priority *Low* or to assign the value *Low* to attribute 'Priority. In either case either Program_Error is raised or the ceiling of the protected object is assigned the value *Low*+1.

Erroneous Execution

{*AI95-00357-01*} {*erroneous execution (cause)* [partial]} If a value of Task_Id is passed as a parameter to any of the subprograms of this package and the corresponding task object no longer exists, the execution of the program is erroneous. 31/2

Documentation Requirements

{*AI95-00357-01*} On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the deadline of a task to be delayed later than what is specified for a single processor. 32/2

Documentation Requirement: Any conditions that cause the completion of the setting of the deadline of a task to be delayed for a multiprocessor. 32.a.1/2

NOTES

18 {*AI95-00357-01*} If two adjacent priority ranges, *A*..*B* and *B*+1..*C* are specified to have policy EDF_Across_Priorities then this is not equivalent to this policy being specified for the single range, *A*..*C*. 33/2

19 {*AI95-00357-01*} The above rules implement the preemption-level protocol (also called Stack Resource Policy protocol) for resource sharing under EDF dispatching. The preemption-level for a task is denoted by its base priority. The definition of a ceiling preemption-level for a protected object follows the existing rules for ceiling locking. 34/2

Implementation Note: {*AI95-00357-01*} An implementation may support additional dispatching policies by replacing absolute deadline with an alternative measure of urgency. 34.a/2

Extensions to Ada 95

{*AI95-00357-01*} {*extensions to Ada 95*} Policy EDF_Across_Priorities and package Dispatching.EDF are new. 34.b/2

D.3 Priority Ceiling Locking

[This clause specifies the interactions between priority task scheduling and protected object ceilings. This interaction is based on the concept of the *ceiling priority* of a protected object.] 1

Syntax

The form of a pragma Locking_Policy is as follows: 2

pragma Locking_Policy(*policy*_identifier); 3

Legality Rules

The *policy*_identifier shall either be Ceiling_Locking or an implementation-defined identifier. 4

Implementation defined: Implementation-defined *policy*_identifiers allowed in a pragma Locking_Policy. 4.a

Post-Compilation Rules

{*configuration pragma (Locking_Policy)* [partial]} {*pragma, configuration (Locking_Policy)* [partial]} A Locking_Policy pragma is a configuration pragma. 5

Dynamic Semantics

{*8652/0073*} {*AI95-00091-01*} {*AI95-00327-01*} {*locking policy*} [A locking policy specifies the details of protected object locking. All protected objects have a priority. The locking policy specifies the meaning of the priority of a protected object, and the relationships between these priorities and task priorities. In addition, the policy specifies the state of a task when it executes a protected action, and how its active priority is affected by the locking.] The *locking policy* is specified by a Locking_Policy pragma. For implementation-defined locking policies, the meaning of the priority of a protected object is 6/2 implementation defined. If no Locking_Policy pragma applies to any of the program units comprising a partition, the locking policy for that partition, as well as the meaning of the priority of a protected object, are implementation defined. {*Priority (of a protected object)*}

6.a/2 **Implementation defined:** The locking policy if no Locking_Policy pragma applies to any unit of a partition.

6.1/2 {*AI95-00327-01*} The expression of a Priority or Interrupt_Priority pragma (see D.1) is evaluated as part of the creation of the corresponding protected object and converted to the subtype System.Any_Priority or System.Interrupt_Priority, respectively. The value of the expression is the initial priority of the corresponding protected object. If no Priority or Interrupt_Priority pragma applies to a protected object, the initial priority is specified by the locking policy. {*implicit subtype conversion (pragma Priority)* [partial]} {*implicit subtype conversion (pragma Interrupt_Priority)* [partial]}

7 There is one predefined locking policy, Ceiling_Locking; this policy is defined as follows:

- 8/2 {*AI95-00327-01*} {*ceiling priority (of a protected object)*} Every protected object has a *ceiling priority*, which is determined by either a Priority or Interrupt_Priority pragma as defined in D.1, or by assignment to the Priority attribute as described in D.5.2. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.
- 9/2 {*AI95-00327-01*} The initial ceiling priority of a protected object is equal to the initial priority for that object.
- 10/2 {*AI95-00327-01*} If an Interrupt_Handler or Attach_Handler pragma (see C.3.1) appears in a protected_definition without an Interrupt_Priority pragma, the initial priority of protected objects of that type is implementation defined, but in the range of the subtype System.Interrupt_Priority.

10.a **Implementation defined:** Default ceiling priorities.

- 11/2 {*AI95-00327-01*} If no pragma Priority, Interrupt_Priority, Interrupt_Handler, or Attach_Handler is specified in the protected_definition, then the initial priority of the corresponding protected object is System.Priority'Last.
- 12 While a task executes a protected action, it inherits the ceiling priority of the corresponding protected object.
- 13 {*Ceiling_Check* [partial]} {*check, language-defined (Ceiling_Check)*} {*Program_Error (raised by failure of run-time check)*} When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; Program_Error is raised if this check fails.

Bounded (Run-Time) Errors

13.1/2 {*AI95-00327-01*} Following any change of priority, it is a bounded error for the active priority of any task with a call queued on an entry of a protected object to be higher than the ceiling priority of the protected object. {*bounded error (cause)* [partial]} In this case one of the following applies:

- 13.2/2 at any time prior to executing the entry body Program_Error is raised in the calling task; {*Program_Error (raised by failure of run-time check)*}
- 13.3/2 when the entry is open the entry body is executed at the ceiling priority of the protected object;
- 13.4/2 when the entry is open the entry body is executed at the ceiling priority of the protected object and then Program_Error is raised in the calling task; or {*Program_Error (raised by failure of run-time check)*}
- 13.5/2 when the entry is open the entry body is executed at the ceiling priority of the protected object that was in effect when the entry call was queued.

13.a.1/2 **Ramification:** Note that the error is “blamed” on the task that did the entry call, not the task that changed the priority of the task or protected object. This seems to make sense for the case of changing the priority of a task blocked on a call, since if the Set_Priority had happened a little bit sooner, before the task queued a call, the entry-calling task would get the error. Similarly, there is no reason not to raise the priority of a task that is executing in an abortable_part, so long as its priority is lowered before it gets to the end and needs to cancel the call. The priority might need to be lowered to allow it to remove the call from the entry queue, in order to avoid violating the ceiling. This seems relatively harmless, since there is an error, and the task is about to start raising an exception anyway.

Implementation Permissions

14 The implementation is allowed to round all ceilings in a certain subrange of System.Priority or System.Interrupt_Priority up to the top of that subrange, uniformly.

14.a **Discussion:** For example, an implementation might use Priority'Last for all ceilings in Priority, and Interrupt_Priority'Last for all ceilings in Interrupt_Priority. This would be equivalent to having two ceiling priorities for protected objects, "nonpreemptible" and "noninterruptible", and is an allowed behavior.

14.b Note that the implementation cannot choose a subrange that crosses the boundary between normal and interrupt priorities.

15/2 {*AI95-00256-01*} Implementations are allowed to define other locking policies, but need not support more than one locking policy per partition.

16 [Since implementations are allowed to place restrictions on code that runs at an interrupt-level active priority (see C.3.1 and D.2.1), the implementation may implement a language feature in terms of a protected object with an implementation-defined ceiling, but the ceiling shall be no less than Priority'Last.]

16.a **Implementation defined:** The ceiling of any protected object used internally by the implementation.

16.b **Proof:** This permission follows from the fact that the implementation can place restrictions on interrupt handlers and on any other code that runs at an interrupt-level active priority.

16.c The implementation might protect a storage pool with a protected object whose ceiling is Priority'Last, which would cause allocators to fail when evaluated at interrupt priority. Note that the ceiling of such an object has to be at least Priority'Last, since there is no permission for allocators to fail when evaluated at a non-interrupt priority.

Implementation Advice

17 The implementation should use names that end with "_Locking" for implementation-defined locking policies.

17.a/2 **Implementation Advice:** Names that end with "_Locking" should be used for implementation-defined locking policies.

NOTES

18 20 While a task executes in a protected action, it can be preempted only by tasks whose active priorities are higher than the ceiling priority of the protected object.

19 21 If a protected object has a ceiling priority in the range of Interrupt_Priority, certain interrupts are blocked while protected actions of that object execute. In the extreme, if the ceiling is Interrupt_Priority'Last, all blockable interrupts are blocked during that time.

20 22 The ceiling priority of a protected object has to be in the Interrupt_Priority range if one of its procedures is to be used as an interrupt handler (see C.3).

21 23 When specifying the ceiling of a protected object, one should choose a value that is at least as high as the highest active priority at which tasks can be executing when they call protected operations of that object. In determining this value the following factors, which can affect active priority, should be considered: the effect of Set_Priority, nested protected operations, entry calls, task activation, and other implementation-defined factors.

22 24 Attaching a protected procedure whose ceiling is below the interrupt hardware priority to an interrupt causes the execution of the program to be erroneous (see C.3.1).

23 25 On a single processor implementation, the ceiling priority rules guarantee that there is no possibility of deadlock involving only protected subprograms (excluding the case where a protected operation calls another protected operation on the same protected object).

Extensions to Ada 95

23.a/2 {*AI95-00327-01*} {*extensions to Ada 95*} All protected objects now have a priority, which is the value of the Priority attribute of D.5.2. How this value is interpreted depends on the locking policy; for instance, the ceiling priority is derived from this value when the locking policy is Ceiling_Locking.

Wording Changes from Ada 95

23.b/2 {*8652/0073*} {*AI95-00091-01*} **Corrigendum:** Corrected the wording to reflect that pragma Locking_Policy cannot be inside of a program unit.

23.c/2 {*AI95-00256-01*} Clarified that an implementation need support only one locking policy (of any kind, language-defined or otherwise) per partition.

23.d/2 {*AI95-00327-01*} The bounded error for the priority of a task being higher than the ceiling of an object it is currently in was moved here from D.5, so that it applies no matter how the situation arises.

D.4 Entry Queuing Policies

1/1 {*8652/0074*} {*AI95-00068-01*} [{*queuing policy*} This clause specifies a mechanism for a user to choose an entry *queuing policy*. It also defines two such policies. Other policies are implementation defined.]

1.a **Implementation defined:** Implementation-defined queuing policies.

Syntax

2 The form of a pragma Queuing_Policy is as follows:

3 **pragma** Queuing_Policy(*policy*_identifier);

Legality Rules

4 The *policy*_identifier shall be either FIFO_Queuing, Priority_Queuing or an implementation-defined identifier.

Post-Compilation Rules

5 {*configuration pragma (Queuing_Policy)* [partial]} {*pragma, configuration (Queuing_Policy)* [partial]} A Queuing_Policy pragma is a configuration pragma.

Dynamic Semantics

6 {*queuing policy*} [A *queuing policy* governs the order in which tasks are queued for entry service, and the order in which different entry queues are considered for service.] The queuing policy is specified by a Queuing_Policy pragma.

6.a **Ramification:** The queuing policy includes entry queuing order, the choice among open alternatives of a selective_accept, and the choice among queued entry calls of a protected object when more than one entry_barrier condition is True.

7/2 {*AI95-00355-01*} Two queuing policies, FIFO_Queuing and Priority_Queuing, are language defined. If no Queuing_Policy pragma applies to any of the program units comprising the partition, the queuing policy for that partition is FIFO_Queuing. The rules for this policy are specified in 9.5.3 and 9.7.1.

8 The Priority_Queuing policy is defined as follows:

9 • {*priority of an entry call*} The calls to an entry [(including a member of an entry family)] are queued in an order consistent with the priorities of the calls. The *priority of an entry call* is initialized from the active priority of the calling task at the time the call is made, but can change later. Within the same priority, the order is consistent with the calling (or requeuing, or priority setting) time (that is, a FIFO order).

- {*8652/0075*} {*AI95-00205-01*} After a call is first queued, changes to the active priority of a task do not affect the priority of the call, unless the base priority of the task is set while the task is blocked on an entry call. 10/1

- When the base priority of a task is set (see D.5), if the task is blocked on an entry call, and the call is queued, the priority of the call is updated to the new active priority of the calling task. This causes the call to be removed from and then reinserted in the queue at the new active priority. 11

  Reason: A task is blocked on an entry call if the entry call is simple, conditional, or timed. If the call came from the triggering_statement of an asynchronous_select, or a requeue thereof, then the task is not blocked on that call; such calls do not have their priority updated. Thus, there can exist many queued calls from a given task (caused by many nested ATC's), but a task can be blocked on only one call at a time. 11.a

  A previous version of Ada 9X required queue reordering in the asynchronous_select case as well. If the call corresponds to a "synchronous" entry call, then the task is blocked while queued, and it makes good sense to move it up in the queue if its priority is raised. 11.b

  However, if the entry call is "asynchronous," that is, it is due to an asynchronous_select whose triggering_statement is an entry call, then the task is not waiting for this entry call, so the placement of the entry call on the queue is irrelevant to the rate at which the task proceeds. 11.c

  Furthermore, when an entry is used for asynchronous_selects, it is almost certain to be a "broadcast" entry or have only one caller at a time. For example, if the entry is used to notify tasks of a mode switch, then all tasks on the entry queue would be signaled when the mode changes. Similarly, if it is indicating some interrupting event such as a control-C, all tasks sensitive to the interrupt will want to be informed that the event occurred. Hence, the order on such a queue is essentially irrelevant. 11.d

  Given the above, it seems an unnecessary semantic and implementation complexity to specify that asynchronous queued calls are moved in response to dynamic priority changes. Furthermore, it is somewhat inconsistent, since the call was originally queued based on the active priority of the task, but dynamic priority changes are changing the base priority of the task, and only indirectly the active priority. We say explicitly that asynchronous queued calls are not affected by normal changes in active priority during the execution of an abortable_part. Saying that, if a change in the base priority affects the active priority, then we do want the calls reordered, would be inconsistent. It would also require the implementation to maintain a readily accessible list of all queued calls which would not otherwise be necessary. 11.e

  Several rules were removed or simplified when we changed the rules so that calls due to asynchronous_selects are never moved due to intervening changes in active priority, be they due to protected actions, some other priority inheritance, or changes in the base priority. 11.f

- When more than one condition of an entry_barrier of a protected object becomes True, and more than one of the respective queues is nonempty, the call with the highest priority is selected. If more than one such call has the same priority, the call that is queued on the entry whose declaration is first in textual order in the protected_definition is selected. For members of the same entry family, the one with the lower family index is selected. 12

- If the expiration time of two or more open delay_alternatives is the same and no other accept_alternatives are open, the sequence_of_statements of the delay_alternative that is first in textual order in the selective_accept is executed. 13

- When more than one alternative of a selective_accept is open and has queued calls, an alternative whose queue has the highest-priority call at its head is selected. If two or more open alternatives have equal-priority queued calls, then a call on the entry in the accept_alternative that is first in textual order in the selective_accept is selected. 14

Implementation Permissions

{*AI95-00256-01*} Implementations are allowed to define other queuing policies, but need not support more than one queuing policy per partition. 15/2

Discussion: {*8652/0116*} {*AI95-00069-01*} {*AI95-00256-01*} This rule is really redundant, as 10.1.5 allows an implementation to limit the use of configuration pragmas to an empty environment. In that case, there would be no way to have multiple policies in a partition. 15.a.1/2

{*AI95-00188-02*} Implementations are allowed to defer the reordering of entry queues following a change of base priority of a task blocked on the entry call if it is not practical to reorder the queue immediately. 15.1/2

15.a.2/2 **Reason:** Priority change is immediate, but the effect of the change on entry queues can be deferred. That is necessary in order to implement priority changes on top of a non-Ada kernel.

15.a.3/2 **Discussion:** The reordering should occur as soon as the blocked task can itself perform the reinsertion into the entry queue.

Implementation Advice

16 The implementation should use names that end with "_Queuing" for implementation-defined queuing policies.

16.a/2 **Implementation Advice:** Names that end with "_Queuing" should be used for implementation-defined queuing policies.

Wording Changes from Ada 95

16.b/2 {*8652/0074*} {*AI95-00068-01*} **Corrigendum:** Corrected the number of queuing policies defined.

16.c/2 {*8652/0075*} {*AI95-00205-01*} **Corrigendum:** Corrected so that a call of Set_Priority in an abortable part does not change the priority of the triggering entry call.

16.d/2 {*AI95-00188-02*} Added a permission to defer queue reordering when the base priority of a task is changed. This is a counterpart to stronger requirements on the implementation of priority change.

16.e/2 {*AI95-00256-01*} Clarified that an implementation need support only one queuing policy (of any kind, language-defined or otherwise) per partition.

16.f/2 {*AI95-00355-01*} Fixed wording to make clear that pragma never appears inside of a unit; rather it "applies to" the unit.

D.5 Dynamic Priorities

1/2 {*AI95-00327-01*} [This clause describes how the priority of an entity can be modified or queried at run time.]

Wording Changes from Ada 95

1.a/2 {*AI95-00327-01*} This clause is turned into two subclauses. This clause introduction is new.

D.5.1 Dynamic Priorities for Tasks

1 [This clause describes how the base priority of a task can be modified or queried at run time.]

Static Semantics

2 The following language-defined library package exists:

3/2
```
{AI95-00362-01} with System;
with Ada.Task_Identification; -- See C.7.1
package Ada.Dynamic_Priorities is
    pragma Preelaborate(Dynamic_Priorities);
```

4
```
    procedure Set_Priority(Priority : in System.Any_Priority;
                           T : in Ada.Task_Identification.Task_Id :=
                           Ada.Task_Identification.Current_Task);
```

5
```
    function Get_Priority (T : Ada.Task_Identification.Task_Id :=
                           Ada.Task_Identification.Current_Task)
                           return System.Any_Priority;
```

6
```
end Ada.Dynamic_Priorities;
```

Dynamic Semantics

7 The procedure Set_Priority sets the base priority of the specified task to the specified Priority value. Set_Priority has no effect if the task is terminated.

The function Get_Priority returns T's current base priority. {*Tasking_Error (raised by failure of run-time check)*} Tasking_Error is raised if the task is terminated. 8

Reason: There is no harm in setting the priority of a terminated task. A previous version of Ada 9X made this a run-time error. However, there is little difference between setting the priority of a terminated task, and setting the priority of a task that is about to terminate very soon; neither case should be an error. Furthermore, the run-time check is not necessarily feasible to implement on all systems, since priority changes might be deferred due to inter-processor communication overhead, so the error might not be detected until after Set_Priority has returned. 8.a

However, we wish to allow implementations to avoid storing "extra" information about terminated tasks. Therefore, we make Get_Priority of a terminated task raise an exception; the implementation need not continue to store the priority of a task that has terminated. 8.b

{*Program_Error (raised by failure of run-time check)*} Program_Error is raised by Set_Priority and Get_Priority if T is equal to Null_Task_Id. 9

{*AI95-00188-02*} On a system with a single processor, the setting of the base priority of a task *T* to the new value occurs immediately at the first point when *T* is outside the execution of a protected action. 10/2

Implementation Note: {*AI95-00188-02*} The priority change is immediate if the target task is on a delay queue or a ready queue outside of a protected action. However, consider when Set_Priority is called by a task T1 to set the priority of T2, if T2 is blocked, waiting on an entry call queued on a protected object, the entry queue needs to be reordered. Since T1 might have a priority that is higher than the ceiling of the protected object, T1 cannot, in general, do the reordering. One way to implement this is to wake T2 up and have T2 do the work. This is similar to the disentangling of queues that needs to happen when a high-priority task aborts a lower-priority task, which might have a call queued on a protected object with a low ceiling. We have an Implementation Permission in D.4 to allow this implementation. We could have required an immediate priority change if on a ready queue during a protected action, but that would have required extra checks for ceiling violations to meet Bounded (Run-Time) Error requirements of D.3 and potentially could cause a protected action to be abandoned in the middle (by raising Program_Error). That seems bad. 10.a/2

Reason: A previous version of Ada 9X made it a run-time error for a high-priority task to set the priority of a lower-priority task that has a queued call on a protected object with a low ceiling. This was changed because: 10.b

- The check was not feasible to implement on all systems, since priority changes might be deferred due to inter-processor communication overhead. The calling task would continue to execute without finding out whether the operation succeeded or not. 10.c
- The run-time check would tend to cause intermittent system failures — how is the caller supposed to know whether the other task happens to have a queued call at any given time? Consider for example an interrupt that needs to trigger a priority change in some task. The interrupt handler could not safely call Set_Priority without knowing exactly what the other task is doing, or without severely restricting the ceilings used in the system. If the interrupt handler wants to hand the job off to a third task whose job is to call Set_Priority, this won't help, because one would normally want the third task to have high priority. 10.d

Bounded (Run-Time) Errors

This paragraph was deleted.{*AI95-00327-01*} 11/2

This paragraph was deleted. 11.a/2

Erroneous Execution

{*erroneous execution (cause)* [partial]} If any subprogram in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous. 12

Ramification: Note that this rule overrides the above rule saying that Program_Error is raised on Get_Priority of a terminated task. If the task object still exists, and the task is terminated, Get_Priority raises Program_Error. However, if the task object no longer exists, calling Get_Priority causes erroneous execution. 12.a

Documentation Requirements

{*AI95-00188-02*} On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the priority of a task to be delayed later than what is specified for a single processor. 12.1/2

Documentation Requirement: Any conditions that cause the completion of the setting of the priority of a task to be delayed for a multiprocessor. 12.a.1/2

Metrics

13 The implementation shall document the following metric:

14
- The execution time of a call to Set_Priority, for the nonpreempting case, in processor clock cycles. This is measured for a call that modifies the priority of a ready task that is not running (which cannot be the calling one), where the new base priority of the affected task is lower than the active priority of the calling task, and the affected task is not on any entry queue and is not executing a protected operation.

14.a/2 **Documentation Requirement:** The metrics for Set_Priority.

NOTES

15/2 26 {*AI95-00321-01*} Setting a task's base priority affects task dispatching. First, it can change the task's active priority. Second, under the FIFO_Within_Priorities policy it always causes the task to move to the tail of the ready queue corresponding to its active priority, even if the new base priority is unchanged.

16 27 Under the priority queuing policy, setting a task's base priority has an effect on a queued entry call if the task is blocked waiting for the call. That is, setting the base priority of a task causes the priority of a queued entry call from that task to be updated and the call to be removed and then reinserted in the entry queue at the new priority (see D.4), unless the call originated from the triggering_statement of an asynchronous_select.

17 28 The effect of two or more Set_Priority calls executed in parallel on the same task is defined as executing these calls in some serial order.

17.a **Proof:** This follows from the general reentrancy requirements stated near the beginning of Annex A, “Predefined Language Environment”.

18 29 The rule for when Tasking_Error is raised for Set_Priority or Get_Priority is different from the rule for when Tasking_Error is raised on an entry call (see 9.5.3). In particular, setting or querying the priority of a completed or an abnormal task is allowed, so long as the task is not yet terminated.

19 30 Changing the priorities of a set of tasks can be performed by a series of calls to Set_Priority for each task separately. For this to work reliably, it should be done within a protected operation that has high enough ceiling priority to guarantee that the operation completes without being preempted by any of the affected tasks.

Extensions to Ada 95

19.a/2 {*AI95-00188-02*} {*extensions to Ada 95*} **Amendment Correction:** Priority changes are now required to be done immediately so long as the target task is not on an entry queue.

19.b/2 {*AI95-00362-01*} Dynamic_Priorities is now Preelaborated, so it can be used in preelaborated units.

Wording Changes from Ada 95

19.c/2 {*AI95-00327-01*} This Ada 95 clause was turned into a subclause. The paragraph numbers are the same as those for D.5 in Ada 95.

19.d/2 {*AI95-00321-01*} There is no “standard” policy anymore, so that phrase was replaced by the name of a specific policy in the notes.

19.e/2 {*AI95-00327-01*} The bounded error for the priority of a task being higher than the ceiling of an object it is currently in was moved to D.3, so that it applies no matter how the situation arises.

D.5.2 Dynamic Priorities for Protected Objects

1/2 {*AI95-00327-01*} This clause specifies how the priority of a protected object can be modified or queried at run time.

Static Semantics

2/2 {*AI95-00327-01*} The following attribute is defined for a prefix P that denotes a protected object:

3/2 P'Priority {*AI95-00327-01*} Denotes a non-aliased component of the protected object P. This component is of type System.Any_Priority and its value is the priority of P. P'Priority denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P.

{*AI95-00327-01*} The initial value of this attribute is the initial value of the priority of the protected object[, and can be changed by an assignment]. 4/2

Dynamic Semantics

{*AI95-00327-01*} If the locking policy Ceiling_Locking (see D.3) is in effect then the ceiling priority of a protected object *P* is set to the value of *P*'Priority at the end of each protected action of *P*. 5/2

{*AI95-00445-01*} If the locking policy Ceiling_Locking is in effect, then for a protected object *P* with either an Attach_Handler or Interrupt_Handler pragma applying to one of its procedures, a check is made that the value to be assigned to *P*'Priority is in the range System.Interrupt_Priority. If the check fails, Program_Error is raised.{*Program_Error (raised by failure of run-time check)*} 6/2

Metrics

{*AI95-00327-01*} The implementation shall document the following metric: 7/2

- The difference in execution time of calls to the following procedures in protected object P: 8/2

9/2
```
protected P is
   procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority);
   procedure Set_Ceiling (Pr : System.Any_Priority);
end P;
```

10/2
```
protected body P is
   procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority) is
   begin
      null;
   end;
   procedure Set_Ceiling (Pr : System.Any_Priority) is
   begin
      P'Priority := Pr;
   end;
end P;
```

Documentation Requirement: The metrics for setting the priority of a protected object. 10.a/2

NOTES
31 {*AI95-00327-01*} Since P'Priority is a normal variable, the value following an assignment to the attribute immediately reflects the new value even though its impact on the ceiling priority of P is postponed until completion of the protected action in which it is executed. 11/2

Extensions to Ada 95

{*AI95-00327-01*} {*AI95-00445-01*} {*extensions to Ada 95*} The ability to dynamically change and query the priority of a protected object is new. 11.a/2

D.6 Preemptive Abort

[This clause specifies requirements on the immediacy with which an aborted construct is completed.] 1

Dynamic Semantics

On a system with a single processor, an aborted construct is completed immediately at the first point that is outside the execution of an abort-deferred operation. 2

Documentation Requirements

On a multiprocessor, the implementation shall document any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. 3

This paragraph was deleted. 3.a/2

Documentation Requirement: On a multiprocessor, any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. 3.b/2

Metrics

4 The implementation shall document the following metrics:

5 • The execution time, in processor clock cycles, that it takes for an abort_statement to cause the completion of the aborted task. This is measured in a situation where a task T2 preempts task T1 and aborts T1. T1 does not have any finalization code. T2 shall verify that T1 has terminated, by means of the Terminated attribute.

6 • On a multiprocessor, an upper bound in seconds, on the time that the completion of an aborted task can be delayed beyond the point that it is required for a single processor.

7/2 • {*AI95-00114-01*} An upper bound on the execution time of an asynchronous_select, in processor clock cycles. This is measured between a point immediately before a task T1 executes a protected operation Pr.Set that makes the condition of an entry_barrier Pr.Wait True, and the point where task T2 resumes execution immediately after an entry call to Pr.Wait in an asynchronous_select. T1 preempts T2 while T2 is executing the abortable part, and then blocks itself so that T2 can execute. The execution time of T1 is measured separately, and subtracted.

8 • An upper bound on the execution time of an asynchronous_select, in the case that no asynchronous transfer of control takes place. This is measured between a point immediately before a task executes the asynchronous_select with a nonnull abortable part, and the point where the task continues execution immediately after it. The execution time of the abortable part is subtracted.

8.a/2 **Documentation Requirement:** The metrics for aborts.

Implementation Advice

9 Even though the abort_statement is included in the list of potentially blocking operations (see 9.5.1), it is recommended that this statement be implemented in a way that never requires the task executing the abort_statement to block.

9.a/2 **Implementation Advice:** The abort_statement should not require the task executing the statement to block.

10 On a multi-processor, the delay associated with aborting a task on another processor should be bounded; the implementation should use periodic polling, if necessary, to achieve this.

10.a/2 **Implementation Advice:** On a multi-processor, the delay associated with aborting a task on another processor should be bounded.

NOTES

11 32 Abortion does not change the active or base priority of the aborted task.

12 33 Abortion cannot be more immediate than is allowed by the rules for deferral of abortion during finalization and in protected actions.

D.7 Tasking Restrictions

1 [This clause defines restrictions that can be used with a pragma Restrictions (see 13.12) to facilitate the construction of highly efficient tasking run-time systems.]

Static Semantics

2 The following *restriction_*identifiers are language defined:

3 {*Restrictions (No_Task_Hierarchy)*} No_Task_Hierarchy All (nonenvironment) tasks depend directly on the environment task of the partition.

4/2 {*8652/0042*} {*AI95-00130-01*} {*AI95-00360-01*} {*Restrictions (No_Nested_Finalization)*} No_Nested_Finalization
Objects of a type that needs finalization (see 7.6) and access types that designate a type that needs finalization shall be declared only at library level.

This paragraph was deleted.{*8652/0042*} {*AI95-00130-01*} 4.a/1

{*Restrictions (No_Abort_Statements)*} No_Abort_Statements There are no abort_statements, and there are no calls on Task_Identification.Abort_Task. 5

{*Restrictions (No_Terminate_Alternatives)*} No_Terminate_Alternatives There are no selective_accepts with terminate_alternatives. 6

{*Restrictions (No_Task_Allocators)*} No_Task_Allocators There are no allocators for task types or types containing task subcomponents. 7

{*Restrictions (No_Implicit_Heap_Allocations)*} No_Implicit_Heap_Allocations There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined. 8

Implementation defined: Any operations that implicitly require heap storage allocation. 8.a

{*AI95-00327-01*} No_Dynamic_Priorities There are no semantic dependences on the package Dynamic_Priorities, and no occurrences of the attribute Priority. {*Restrictions (No_Dynamic_Priorities)*} 9/2

{*AI95-00305-01*} {*AI95-00394-01*} {*Restrictions (No_Dynamic_Attachment)*} No_Dynamic_Attachment There is no call to any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference). 10/2

{*AI95-00305-01*} {*Restrictions (No_Local_Protected_Objects)*} No_Local_Protected_Objects Protected objects shall be declared only at library level. 10.1/2

{*AI95-00297-01*} {*Restrictions (No_Local_Timing_Events)*} No_Local_Timing_Events Timing_Events shall be declared only at library level. 10.2/2

{*AI95-00305-01*} {*Restrictions (No_Protected_Type_Allocators)*} No_Protected_Type_Allocators There are no allocators for protected types or types containing protected type subcomponents. 10.3/2

{*AI95-00305-01*} {*Restrictions (No_Relative_Delay)*} No_Relative_Delay There are no delay_relative_statements. 10.4/2

{*AI95-00305-01*} {*Restrictions (No_Requeue_Statements)*} No_Requeue_Statements There are no requeue_statements. 10.5/2

{*AI95-00305-01*} {*Restrictions (No_Select_Statements)*} No_Select_Statements There are no select_statements. 10.6/2

{*AI95-00394-01*} {*Restrictions (No_Specific_Termination_Handlers)*} No_Specific_Termination_Handlers There are no calls to the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination. 10.7/2

{*AI95-00305-01*} {*Restrictions (Simple_Barriers)*} Simple_Barriers The Boolean expression in an entry barrier shall be either a static Boolean expression or a Boolean component of the enclosing protected object. 10.8/2

The following *restriction_parameter*_identifiers are language defined: 11

{*Restrictions (Max_Select_Alternatives)*} Max_Select_Alternatives Specifies the maximum number of alternatives in a selective_accept. 12

{*Restrictions (Max_Task_Entries)*} Max_Task_Entries Specifies the maximum number of entries per task. The bounds of every entry family of a task unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static. [A value of zero indicates that no rendezvous are possible.] 13

14 Max_Protected_Entries
Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static. {*Restrictions (Max_Protected_Entries)*}

Dynamic Semantics

15/2 {*8652/0076*} {*AI95-00067-01*} {*AI95-00305-01*} The following *restriction_*identifier is language defined:

15.1/2 {*AI95-00305-01*} {*AI95-00394-01*} {*Restrictions (No_Task_Termination)*} No_Task_Termination
All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate. If there is a fall-back handler (see C.7.3) set for the partition it should be called when the first task attempts to terminate.

15.a.1/2 **Implementation defined:** When restriction No_Task_Termination applies to a partition, what happens when a task terminates.

16 The following *restriction_parameter_*identifiers are language defined:

17/1 {*8652/0076*} {*AI95-00067-01*} {*Restrictions (Max_Storage_At_Blocking)*} Max_Storage_At_Blocking
Specifies the maximum portion [(in storage elements)] of a task's Storage_Size that can be retained by a blocked task. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; {*Storage_Check* [partial]} {*check, language-defined (Storage_Check)*} {*Storage_Error (raised by failure of run-time check)*} otherwise, the behavior is implementation defined.

17.a.1/2 **Implementation defined:** The behavior when restriction Max_Storage_At_Blocking is violated.

18/1 {*8652/0076*} {*AI95-00067-01*} {*Restrictions (Max_Asynchronous_Select_Nesting)*} Max_Asynchronous_Select_Nesting
Specifies the maximum dynamic nesting level of asynchronous_selects. A value of zero prevents the use of any asynchronous_select and, if a program contains an asynchronous_-select, it is illegal. If an implementation chooses to detect a violation of this restriction for values other than zero, Storage_Error should be raised; {*Storage_Check* [partial]} {*check, language-defined (Storage_Check)*} {*Storage_Error (raised by failure of run-time check)*} otherwise, the behavior is implementation defined.

18.a.1/2 **Implementation defined:** The behavior when restriction Max_Asynchronous_Select_Nesting is violated.

19/1 {*8652/0076*} {*AI95-00067-01*} {*Restrictions (Max_Tasks)*} Max_Tasks
Specifies the maximum number of task creations that may be executed over the lifetime of a partition, not counting the creation of the environment task. A value of zero prevents any task creation and, if a program contains a task creation, it is illegal. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; {*Storage_Check* [partial]} {*check, language-defined (Storage_Check)*} {*Storage_Error (raised by failure of run-time check)*} otherwise, the behavior is implementation defined.

19.a **Ramification:** Note that this is not a limit on the number of tasks active at a given time; it is a limit on the total number of task creations that occur.

19.b **Implementation Note:** We envision an implementation approach that places TCBs or pointers to them in a fixed-size table, and never reuses table elements.

19.b.1/2 **Implementation defined:** The behavior when restriction Max_Tasks is violated.

19.1/2 {*AI95-00305-01*} {*Restrictions (Max_Entry_Queue_Length)*} Max_Entry_Queue_Length
Max_Entry_Queue_Length defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error at the point of the call or requeue.{*Program_Error (raised by failure of run-time check)*}

It is implementation defined whether the use of pragma Restrictions results in a reduction in executable program size, storage requirements, or execution time. If possible, the implementation should provide quantitative descriptions of such effects for each restriction. 20

> **Implementation defined:** Whether the use of pragma Restrictions results in a reduction in program code or data size or execution time. 20.a/2

Implementation Advice

When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation. 21

> **Implementation Advice:** When feasible, specified restrictions should be used to produce a more efficient implementation. 21.a/2

NOTES
34 The above Storage_Checks can be suppressed with pragma Suppress. 22

Incompatibilities With Ada 95

> {*AI95-00360-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** The No_Nested_Finalization is now defined in terms of types that need finalization. These types include a variety of language-defined types that *might* be implemented with a controlled type. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and one of these language-defined types does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as other Ada compilers may have had a controlled part, and thus would be illegal under the restriction. 22.a/2

Extensions to Ada 95

> {*AI95-00297-01*} {*AI95-00305-01*} {*AI95-00394-01*} {*extensions to Ada 95*} Restrictions No_Dynamic_Attachment, No_Local_Protected_Objects, No_Protected_Type_Allocators, No_Local_Timing_Events, No_Relative_Delay, No_Requeue_Statement, No_Select_Statements, No_Specific_Termination_Handlers, No_Task_Termination, Max_Entry_Queue_Length, and Simple_Barriers are newly added to Ada. 22.b/2

Wording Changes from Ada 95

> {*8652/0042*} {*AI95-00130-01*} **Corrigendum:** Clarified that No_Nested_Finalization covered task and protected parts as well. 22.c/2

> {*8652/0076*} {*AI95-00067-01*} **Corrigendum:** Changed the description of Max_Tasks and Max_Asynchronous_Select_Nested to eliminate conflicts with the High Integrity Annex (see H.4). 22.d/2

> {*AI95-00327-01*} Added using of the new Priority attribute to the restriction No_Dynamic_Priorities. 22.e/2

> {*AI95-00394-01*} Restriction No_Asynchronous_Control is now obsolescent. 22.f/2

D.8 Monotonic Time

[This clause specifies a high-resolution, monotonic clock package.] 1

Static Semantics

The following language-defined library package exists: 2

```
package Ada.Real_Time is                                                  3

  type Time is private;                                                   4
  Time_First : constant Time;
  Time_Last : constant Time;
  Time_Unit : constant := implementation-defined-real-number;

  type Time_Span is private;                                              5
  Time_Span_First : constant Time_Span;
  Time_Span_Last : constant Time_Span;
  Time_Span_Zero : constant Time_Span;
  Time_Span_Unit : constant Time_Span;

  Tick : constant Time_Span;                                              6
  function Clock return Time;
```

7
```
      function "+" (Left : Time; Right : Time_Span) return Time;
      function "+" (Left : Time_Span; Right : Time) return Time;
      function "-" (Left : Time; Right : Time_Span) return Time;
      function "-" (Left : Time; Right : Time) return Time_Span;
```
8
```
      function "<" (Left, Right : Time) return Boolean;
      function "<="(Left, Right : Time) return Boolean;
      function ">" (Left, Right : Time) return Boolean;
      function ">="(Left, Right : Time) return Boolean;
```
9
```
      function "+" (Left, Right : Time_Span) return Time_Span;
      function "-" (Left, Right : Time_Span) return Time_Span;
      function "-" (Right : Time_Span) return Time_Span;
      function "*" (Left : Time_Span; Right : Integer) return Time_Span;
      function "*" (Left : Integer; Right : Time_Span) return Time_Span;
      function "/" (Left, Right : Time_Span) return Integer;
      function "/" (Left : Time_Span; Right : Integer) return Time_Span;
```
10
```
      function "abs"(Right : Time_Span) return Time_Span;
```
11/1 *This paragraph was deleted.*

12
```
      function "<" (Left, Right : Time_Span) return Boolean;
      function "<="(Left, Right : Time_Span) return Boolean;
      function ">" (Left, Right : Time_Span) return Boolean;
      function ">="(Left, Right : Time_Span) return Boolean;
```
13
```
      function To_Duration (TS : Time_Span) return Duration;
      function To_Time_Span (D : Duration) return Time_Span;
```
14/2
```
    {AI95-00386-01}   function Nanoseconds  (NS : Integer) return Time_Span;
      function Microseconds (US : Integer) return Time_Span;
      function Milliseconds (MS : Integer) return Time_Span;
      function Seconds      (S  : Integer) return Time_Span;
      function Minutes      (M  : Integer) return Time_Span;
```
15
```
      type Seconds_Count is range implementation-defined;
```
16
```
      procedure Split(T : in Time; SC : out Seconds_Count; TS : out Time_Span);
      function Time_Of(SC : Seconds_Count; TS : Time_Span) return Time;
```
17
```
    private
       ... -- not specified by the language
    end Ada.Real_Time;
```
17.a/2 *This paragraph was deleted.*

18 {*real time*} In this Annex, *real time* is defined to be the physical time as observed in the external environment. The type Time is a *time type* as defined by 9.6; [values of this type may be used in a delay_until_statement.] Values of this type represent segments of an ideal time line. The set of values of the type Time corresponds one-to-one with an implementation-defined range of mathematical integers.

18.a **Discussion:** Informally, real time is defined to be the International Atomic Time (TAI) which is monotonic and nondecreasing. We use it here for the purpose of discussing rate of change and monotonic behavior only. It does not imply anything about the absolute value of Real_Time.Clock, or about Real_Time.Time being synchronized with TAI. It is also used for real time in the metrics, for comparison purposes.

18.b **Implementation Note:** The specification of TAI as "real time" does not preclude the use of a simulated TAI clock for simulated execution environments.

19 {*epoch*} {*unspecified* [partial]} The Time value I represents the half-open real time interval that starts with E+I*Time_Unit and is limited by E+(I+1)*Time_Unit, where Time_Unit is an implementation-defined real number and E is an unspecified origin point, the *epoch*, that is the same for all values of the type Time. It is not specified by the language whether the time values are synchronized with any standard time reference. [For example, E can correspond to the time of system initialization or it can correspond to the epoch of some time standard.]

19.a **Discussion:** E itself does not have to be a proper time value.

19.b This half-open interval I consists of all real numbers R such that E+I*Time_Unit <= R < E+(I+1)*Time_Unit.

20 Values of the type Time_Span represent length of real time duration. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers. The Time_Span value corresponding to the integer I represents the real-time duration I*Time_Unit.

20.a **Reason:** The purpose of this type is similar to Standard.Duration; the idea is to have a type with a higher resolution.

20.b **Discussion:** We looked at many possible names for this type: Real_Time.Duration, Fine_Duration, Interval, Time_Interval_Length, Time_Measure, and more. Each of these names had some problems, and we've finally settled for Time_Span.

21 Time_First and Time_Last are the smallest and largest values of the Time type, respectively. Similarly, Time_Span_First and Time_Span_Last are the smallest and largest values of the Time_Span type, respectively.

22 A value of type Seconds_Count represents an elapsed time, measured in seconds, since the epoch.

Dynamic Semantics

23 Time_Unit is the smallest amount of real time representable by the Time type; it is expressed in seconds. Time_Span_Unit is the difference between two successive values of the Time type. It is also the smallest positive value of type Time_Span. Time_Unit and Time_Span_Unit represent the same real time duration. {*clock tick*} A *clock tick* is a real time interval during which the clock value (as observed by calling the Clock function) remains constant. Tick is the average length of such intervals.

24/2 {*AI95-00432-01*} The function To_Duration converts the value TS to a value of type Duration. Similarly, the function To_Time_Span converts the value D to a value of type Time_Span. For To_Duration, the result is rounded to the nearest value of type Duration (away from zero if exactly halfway between two values). If the result is outside the range of Duration, Constraint_Error is raised. For To_Time_Span, the value of D is first rounded to the nearest integral multiple of Time_Unit, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of Time_Span, Constraint_Error is raised. Otherwise, the value is converted to the type Time_Span.

25 To_Duration(Time_Span_Zero) returns 0.0, and To_Time_Span(0.0) returns Time_Span_Zero.

26/2 {*AI95-00386-01*} {*AI95-00432-01*} The functions Nanoseconds, Microseconds, Milliseconds, Seconds, and Minutes convert the input parameter to a value of the type Time_Span. NS, US, MS, S, and M are interpreted as a number of nanoseconds, microseconds, milliseconds, seconds, and minutes respectively. The input parameter is first converted to seconds and rounded to the nearest integral multiple of Time_Unit, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of Time_Span, Constraint_Error is raised. Otherwise, the rounded value is converted to the type Time_Span.

26.a/2 *This paragraph was deleted.*{*AI95-00432-01*}

27 The effects of the operators on Time and Time_Span are as for the operators defined for integer types.

27.a **Implementation Note:** Though time values are modeled by integers, the types Time and Time_Span need not be implemented as integers.

28 The function Clock returns the amount of time since the epoch.

29 The effects of the Split and Time_Of operations are defined as follows, treating values of type Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split(T,SC,TS) is to set SC and TS to values such that T*Time_Unit = SC*1.0 + TS*Time_Unit, and 0.0 <= TS*Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the value T such that T*Time_Unit = SC*1.0 + TS*Time_Unit.

Implementation Requirements

30 The range of Time values shall be sufficient to uniquely represent the range of real times from program start-up to 50 years later. Tick shall be no greater than 1 millisecond. Time_Unit shall be less than or equal to 20 microseconds.

30.a **Implementation Note:** The required range and accuracy of Time are such that 32-bits worth of seconds and 32-bits worth of ticks in a second could be used as the representation.

31 Time_Span_First shall be no greater than –3600 seconds, and Time_Span_Last shall be no less than 3600 seconds.

31.a **Reason:** This is equivalent to ± one hour and there is still room for a two-microsecond resolution.

32 {*clock jump*} A *clock jump* is the difference between two successive distinct values of the clock (as observed by calling the Clock function). There shall be no backward clock jumps.

Documentation Requirements

33 The implementation shall document the values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick.

33.a/2 **Documentation Requirement:** The values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick for package Real_Time.

34 The implementation shall document the properties of the underlying time base used for the clock and for type Time, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

34.a.1/2 **Documentation Requirement:** The properties of the underlying time base used in package Real_Time.

34.a **Discussion:** If there is an underlying operating system, this might include information about which system call is used to implement the clock. Otherwise, it might include information about which hardware clock is used.

35 The implementation shall document whether or not there is any synchronization with external time references, and if such synchronization exists, the sources of synchronization information, the frequency of synchronization, and the synchronization method applied.

35.a.1/2 **Documentation Requirement:** Any synchronization of package Real_Time with external time references.

36/1 The implementation shall document any aspects of the external environment that could interfere with the clock behavior as defined in this clause.

36.a.1/2 **Documentation Requirement:** Any aspects of the external environment that could interfere with package Real_Time.

36.a **Discussion:** For example, the implementation is allowed to rely on the time services of an underlying operating system, and this operating system clock can implement time zones or allow the clock to be reset by an operator. This dependence has to be documented.

Metrics

37 For the purpose of the metrics defined in this clause, real time is defined to be the International Atomic Time (TAI).

38 The implementation shall document the following metrics:

39 • An upper bound on the real-time duration of a clock tick. This is a value D such that if t1 and t2 are any real times such that t1 < t2 and $\text{Clock}_{t1} = \text{Clock}_{t2}$ then t2 – t1 <= D.

40 • An upper bound on the size of a clock jump.

41 • {*drift rate*} An upper bound on the *drift rate* of Clock with respect to real time. This is a real number D such that

42 $E*(1–D) <= (\text{Clock}_{t+E} – \text{Clock}_t) <= E*(1+D)$
provided that: $\text{Clock}_t + E*(1+D) <= \text{Time\_Last}$.

- where $Clock_t$ is the value of Clock at time t, and E is a real time duration not less than 24 hours. The value of E used for this metric shall be reported. 43

Reason: This metric is intended to provide a measurement of the long term (cumulative) deviation; therefore, 24 hours is the lower bound on the measurement period. On some implementations, this is also the maximum period, since the language does not require that the range of the type Duration be more than 24 hours. On those implementations that support longer-range Duration, longer measurements should be performed. 43.a

- An upper bound on the execution time of a call to the Clock function, in processor clock cycles. 44
- Upper bounds on the execution times of the operators of the types Time and Time_Span, in processor clock cycles. 45

Implementation Note: A fast implementation of the Clock function involves repeated reading until you get the same value twice. It is highly improbable that more than three reads will be necessary. Arithmetic on time values should not be significantly slower than 64-bit arithmetic in the underlying machine instruction set. 45.a

Documentation Requirement: The metrics for package Real_Time. 45.a.1/2

Implementation Permissions

Implementations targeted to machines with word size smaller than 32 bits need not support the full range and granularity of the Time and Time_Span types. 46

Discussion: These requirements are based on machines with a word size of 32 bits. 46.a

Since the range and granularity are implementation defined, the supported values need to be documented. 46.b

Implementation Advice

When appropriate, implementations should provide configuration mechanisms to change the value of Tick. 47

Implementation Advice: When appropriate, mechanisms to change the value of Tick should be provided. 47.a.1/2

Reason: This is often needed when the compilation system was originally targeted to a particular processor with a particular interval timer, but the customer uses the same processor with a different interval timer. 47.a

Discussion: Tick is a deferred constant and not a named number specifically for this purpose. 47.b

Implementation Note: This can be achieved either by pre-run-time configuration tools, or by having Tick be initialized (in the package private part) by a function call residing in a board specific module. 47.c

It is recommended that Calendar.Clock and Real_Time.Clock be implemented as transformations of the same time base. 48

Implementation Advice: Calendar.Clock and Real_Time.Clock should be transformations of the same time base. 48.a.1/2

It is recommended that the “best” time base which exists in the underlying system be available to the application through Clock. “Best” may mean highest accuracy or largest range. 49

Implementation Advice: The “best” time base which exists in the underlying system should be available to the application through Real_Time.Clock. 49.a.1/2

NOTES

35 The rules in this clause do not imply that the implementation can protect the user from operator or installation errors which could result in the clock being set incorrectly. 50

36 Time_Unit is the granularity of the Time type. In contrast, Tick represents the granularity of Real_Time.Clock. There is no requirement that these be the same. 51

Incompatibilities With Ada 95

{*AI95-00386-01*} {*incompatibilities with Ada 95*} Functions Seconds and Minutes are newly added to Real_Time. If Real_Time is referenced in a use_clause, and an entity *E* with a defining_identifier of Seconds or Minutes is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 51.a/2

Wording Changes from Ada 95

51.b/2 {*AI95-00432-01*} Added wording explaining how and when many of these functions can raise Constraint_Error. While there always was an intent to raise Constraint_Error if the values did not fit, there never was any wording to that effect, and since Time_Span was a private type, the normal numeric type rules do not apply to it.

D.9 Delay Accuracy

1 [This clause specifies performance requirements for the delay_statement. The rules apply both to delay_relative_statement and to delay_until_statement. Similarly, they apply equally to a simple delay_statement and to one which appears in a delay_alternative.]

Dynamic Semantics

2 The effect of the delay_statement for Real_Time.Time is defined in terms of Real_Time.Clock:

3 • If C_1 is a value of Clock read before a task executes a delay_relative_statement with duration D, and C_2 is a value of Clock read after the task resumes execution following that delay_statement, then $C_2 – C_1$ >= D.

4 • If C is a value of Clock read after a task resumes execution following a delay_until_statement with Real_Time.Time value T, then C >= T.

5 {*potentially blocking operation (delay_statement)* [partial]} {*blocking, potentially (delay_statement)* [partial]} A simple delay_statement with a negative or zero value for the expiration time does not cause the calling task to be blocked; it is nevertheless a potentially blocking operation (see 9.5.1).

6/2 When a delay_statement appears in a delay_alternative of a timed_entry_call the selection of the entry call is attempted, regardless of the specified expiration time. When a delay_statement appears in a select_alternative, and a call is queued on one of the open entries, the selection of that entry call proceeds, regardless of the value of the delay expression.

6.a **Ramification:** The effect of these requirements is that one has to always attempt a rendezvous, regardless of the value of the delay expression. This can be tested by issuing a timed_entry_call with an expiration time of zero, to an open entry.

Documentation Requirements

7 The implementation shall document the minimum value of the delay expression of a delay_relative_statement that causes the task to actually be blocked.

7.a/2 **Documentation Requirement:** The minimum value of the delay expression of a delay_relative_statement that causes a task to actually be blocked.

8 The implementation shall document the minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked.

8.a/2 *This paragraph was deleted.*

8.b/2 **Documentation Requirement:** The minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked.

Metrics

9 The implementation shall document the following metrics:

10 • An upper bound on the execution time, in processor clock cycles, of a delay_relative_statement whose requested value of the delay expression is less than or equal to zero.

11 • An upper bound on the execution time, in processor clock cycles, of a delay_until_statement whose requested value of the delay expression is less than or equal to the value of Real_Time.Clock at the time of executing the statement. Similarly, for Calendar.Clock.

- {*lateness*} {*actual duration*} An upper bound on the *lateness* of a delay_relative_statement, for a positive value of the delay expression, in a situation where the task has sufficient priority to preempt the processor as soon as it becomes ready, and does not need to wait for any other execution resources. The upper bound is expressed as a function of the value of the delay expression. The lateness is obtained by subtracting the value of the delay expression from the *actual duration*. The actual duration is measured from a point immediately before a task executes the delay_statement to a point immediately after the task resumes execution following this statement. 12
- An upper bound on the lateness of a delay_until_statement, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it does not need to wait for any other execution resources. The upper bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a delay_until_statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement. 13

Documentation Requirement: The metrics for delay statements. 13.a/2

NOTES

This paragraph was deleted.{*AI95-00355-01*} 14/2

Wording Changes from Ada 83

The rules regarding a timed_entry_call with a very small positive Duration value, have been tightened to always require the check whether the rendezvous is immediately possible. 14.a

Wording Changes from Ada 95

{*AI95-00355-01*} The note about "voluntary round-robin', while still true, has been deleted as potentially confusing as it is describing a different kind of round-robin than is defined by the round-robin dispatching policy. 14.b/2

D.10 Synchronous Task Control

[This clause describes a language-defined private semaphore (suspension object), which can be used for *two-stage suspend* operations and as a simple building block for implementing higher-level queues.] 1

Static Semantics

The following language-defined package exists: 2

```
{AI95-00362-01} package Ada.Synchronous_Task_Control is                3/2
  pragma Preelaborate(Synchronous_Task_Control);

  type Suspension_Object is limited private;                           4
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object);
private
     ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

The type Suspension_Object is a by-reference type. 5

Implementation Note: {*AI95-00318-02*} The implementation can ensure this by, for example, making the full view an explicitly limited record type. 5.a/2

Dynamic Semantics

{*AI95-00114-01*} An object of the type Suspension_Object has two visible states: True and False. Upon initialization, its value is set to False. 6/2

6.a **Discussion:** This object is assumed to be private to the declaring task, i.e. only that task will call Suspend_Until_True on this object, and the count of callers is at most one. Other tasks can, of course, change and query the state of this object.

7/2 {*AI95-00114-01*} The operations Set_True and Set_False are atomic with respect to each other and with respect to Suspend_Until_True; they set the state to True and False respectively.

8 Current_State returns the current state of the object.

8.a **Discussion:** This state can change immediately after the operation returns.

9/2 {*AI95-00114-01*} The procedure Suspend_Until_True blocks the calling task until the state of the object S is True; at that point the task becomes ready and the state of the object becomes False.

10 {*potentially blocking operation (Suspend_Until_True)* [partial]} {*blocking, potentially (Suspend_Until_True)* [partial]} {*Program_Error (raised by failure of run-time check)*} Program_Error is raised upon calling Suspend_Until_True if another task is already waiting on that suspension object. Suspend_Until_True is a potentially blocking operation (see 9.5.1).

Implementation Requirements

11 The implementation is required to allow the calling of Set_False and Set_True during any protected action, even one that has its ceiling priority in the Interrupt_Priority range.

Extensions to Ada 95

11.a/2 {*AI95-00362-01*} {*extensions to Ada 95*} Synchronous_Task_Control is now Preelaborated, so it can be used in preelaborated units.

D.11 Asynchronous Task Control

1 [This clause introduces a language-defined package to do asynchronous suspend/resume on tasks. It uses a conceptual *held priority* value to represent the task's *held* state.]

Static Semantics

2 The following language-defined library package exists:

3/2
```
{AI95-00362-01} with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
  pragma Preelaborate(Asynchronous_Task_Control);
  procedure Hold(T : in Ada.Task_Identification.Task_Id);
  procedure Continue(T : in Ada.Task_Identification.Task_Id);
  function Is_Held(T : Ada.Task_Identification.Task_Id)
   return Boolean;
end Ada.Asynchronous_Task_Control;
```

Dynamic Semantics

4/2 {*AI95-00357-01*} {*task state (held)* [partial]} {*held priority*} {*idle task*} After the Hold operation has been applied to a task, the task becomes *held*. For each processor there is a conceptual *idle task*, which is always ready. The base priority of the idle task is below System.Any_Priority'First. The *held priority* is a constant of the type Integer whose value is below the base priority of the idle task.

4.a **Discussion:** The held state should not be confused with the blocked state as defined in 9.2; the task is still ready.

4.1/2 {*AI95-00357-01*} For any priority below System.Any_Priority'First, the task dispatching policy is FIFO_Within_Priorities.

4.b/2 **To be honest:** This applies even if a Task_Dispatching_Policy specifies the policy for all of the priorities of the partition.

4.c/2 **Ramification:** A task at the held priority never runs, so it is not necessary to implement FIFO_Within_Priorities for systems that have only one policy (such as EDF_Across_Priorities).

{*AI95-00357-01*} The Hold operation sets the state of T to held. For a held task, the active priority is reevaluated as if the base priority of the task were the held priority. 5/2

Ramification: For example, if T is currently inheriting priorities from other sources (e.g. it is executing in a protected action), its active priority does not change, and it continues to execute until it leaves the protected action. 5.a

{*AI95-00357-01*} The Continue operation resets the state of T to not-held; its active priority is then reevaluated as determined by the task dispatching policy associated with its base priority. 6/2

The Is_Held function returns True if and only if T is in the held state. 7

Discussion: Note that the state of T can be changed immediately after Is_Held returns. 7.a

As part of these operations, a check is made that the task identified by T is not terminated. {*Tasking_Error (raised by failure of run-time check)*} Tasking_Error is raised if the check fails. {*Program_Error (raised by failure of run-time check)*} Program_Error is raised if the value of T is Null_Task_Id. 8

Erroneous Execution

{*erroneous execution (cause)* [partial]} If any operation in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous. 9

Implementation Permissions

An implementation need not support Asynchronous_Task_Control if it is infeasible to support it in the target environment. 10

Reason: A direct implementation of the Asynchronous_Task_Control semantics using priorities is not necessarily efficient enough. Thus, we envision implementations that use some other mechanism to set the "held" state. If there is no other such mechanism, support for Asynchronous_Task_Control might be infeasible, because an implementation in terms of priority would require one idle task per processor. On some systems, programs are not supposed to know how many processors are available, so creating enough idle tasks would be problematic. 10.a

NOTES

37 It is a consequence of the priority rules that held tasks cannot be dispatched on any processor in a partition (unless they are inheriting priorities) since their priorities are defined to be below the priority of any idle task. 11

38 The effect of calling Get_Priority and Set_Priority on a Held task is the same as on any other task. 12

39 Calling Hold on a held task or Continue on a non-held task has no effect. 13

40 The rules affecting queuing are derived from the above rules, in addition to the normal priority rules: 14

- When a held task is on the ready queue, its priority is so low as to never reach the top of the queue as long as there are other tasks on that queue. 15
- If a task is executing in a protected action, inside a rendezvous, or is inheriting priorities from other sources (e.g. when activated), it continues to execute until it is no longer executing the corresponding construct. 16
- If a task becomes held while waiting (as a caller) for a rendezvous to complete, the active priority of the accepting task is not affected. 17
- {*8652/0077*} {*AI95-00111-01*} If a task becomes held while waiting in a selective_accept, and an entry call is issued to one of the open entries, the corresponding accept_alternative executes. When the rendezvous completes, the active priority of the accepting task is lowered to the held priority (unless it is still inheriting from other sources), and the task does not execute until another Continue. 18/1
- The same holds if the held task is the only task on a protected entry queue whose barrier becomes open. The corresponding entry body executes. 19

Extensions to Ada 95

{*AI95-00362-01*} {*extensions to Ada 95*} Asynchronous_Task_Control is now Preelaborated, so it can be used in preelaborated units. 19.a/2

Wording Changes from Ada 95

{*8652/0077*} {*AI95-00111-01*} **Corrigendum:** Corrected to eliminate the use of the undefined term "accept body". 19.b/2

19.c/2 {*AI95-00357-01*} The description of held tasks was changed to reflect that the calculation of active priorities depends on the dispatching policy of the base priority. Thus, the policy of the held priority was specified in order to avoid surprises (especially when using the EDF policy).

D.12 Other Optimizations and Determinism Rules

1 [This clause describes various requirements for improving the response and determinism in a real-time system.]

Implementation Requirements

2 If the implementation blocks interrupts (see C.3) not as a result of direct user action (e.g. an execution of a protected action) there shall be an upper bound on the duration of this blocking.

2.a **Ramification:** The implementation shall not allow itself to be interrupted when it is in a state where it is unable to support all the language-defined operations permitted in the execution of interrupt handlers. (see 9.5.1).

3 The implementation shall recognize entry-less protected types. The overhead of acquiring the execution resource of an object of such a type (see 9.5.1) shall be minimized. In particular, there should not be any overhead due to evaluating entry_barrier conditions.

3.a **Implementation Note:** Ideally the overhead should just be a spin-lock.

4 Unchecked_Deallocation shall be supported for terminated tasks that are designated by access types, and shall have the effect of releasing all the storage associated with the task. This includes any run-time system or heap storage that has been implicitly allocated for the task by the implementation.

Documentation Requirements

5 The implementation shall document the upper bound on the duration of interrupt blocking caused by the implementation. If this is different for different interrupts or interrupt priority levels, it should be documented for each case.

5.a/2 *This paragraph was deleted.*

5.b/2 **Documentation Requirement:** The upper bound on the duration of interrupt blocking caused by the implementation.

Metrics

6 The implementation shall document the following metric:

7 • The overhead associated with obtaining a mutual-exclusive access to an entry-less protected object. This shall be measured in the following way:

8 For a protected object of the form:

9
```
protected Lock is
   procedure Set;
   function Read return Boolean;
private
   Flag : Boolean := False;
end Lock;
```

10
```
protected body Lock is
   procedure Set is
   begin
      Flag := True;
   end Set;
   function Read return Boolean
   Begin
      return Flag;
   end Read;
end Lock;
```

11 The execution time, in processor clock cycles, of a call to Set. This shall be measured between the point just before issuing the call, and the point just after the call completes. The function Read shall be called later to verify that Set was indeed called (and not optimized away). The calling task shall have sufficiently high priority as to not be preempted during the measurement period. The protected object shall have sufficiently high ceiling priority to allow the task to call Set.

12 For a multiprocessor, if supported, the metric shall be reported for the case where no contention (on the execution resource) exists [from tasks executing on other processors].

12.a/2 **Documentation Requirement:** The metrics for entry-less protected objects.

D.13 Run-time Profiles

1/2 {*AI95-00249-01*} [This clause specifies a mechanism for defining run-time profiles.]

Syntax

2/2 {*AI95-00249-01*} The form of a pragma Profile is as follows:

3/2 **pragma** Profile (*profile*_identifier {, *profile*_pragma_argument_association});

Legality Rules

4/2 {*AI95-00249-01*} The *profile*_identifier shall be the name of a run-time profile. The semantics of any *profile*_pragma_argument_associations are defined by the run-time profile specified by the *profile*_identifier.

Static Semantics

5/2 {*AI95-00249-01*} A profile is equivalent to the set of configuration pragmas that is defined for each run-time profile.

Post-Compilation Rules

6/2 {*AI95-00249-01*} {*configuration pragma (Profile)* [partial]} {*pragma, configuration (Profile)* [partial]} A pragma Profile is a configuration pragma. There may be more than one pragma Profile for a partition.

Extensions to Ada 95

6.a/2 {*AI95-00249-01*} {*extensions to Ada 95*} Pragma Profile is new.

D.13.1 The Ravenscar Profile

1/2 {*AI95-00249-01*} [This clause defines the Ravenscar profile.]{*Ravenscar*}

Legality Rules

2/2 {*AI95-00249-01*} The *profile*_identifier Ravenscar is a run-time profile. For run-time profile Ravenscar, there shall be no *profile*_pragma_argument_associations.

Static Semantics

3/2 {*AI95-00249-01*} The run-time profile Ravenscar is equivalent to the following set of pragmas:

4/2
```
{AI95-00249-01} {AI95-00297-01} {AI95-00394-01} pragma Task_Dispatching_Policy
(FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
                No_Abort_Statements,
                No_Dynamic_Attachment,
                No_Dynamic_Priorities,
                No_Implicit_Heap_Allocations,
                No_Local_Protected_Objects,
                No_Local_Timing_Events,
                No_Protected_Type_Allocators,
                No_Relative_Delay,
                No_Requeue_Statements,
                No_Select_Statements,
                No_Specific_Termination_Handlers,
                No_Task_Allocators,
                No_Task_Hierarchy,
                No_Task_Termination,
                Simple_Barriers,
                Max_Entry_Queue_Length => 1,
                Max_Protected_Entries => 1,
                Max_Task_Entries => 0,
                No_Dependence => Ada.Asynchronous_Task_Control,
                No_Dependence => Ada.Calendar,
                No_Dependence => Ada.Execution_Time.Group_Budget,
                No_Dependence => Ada.Execution_Time.Timers,
                No_Dependence => Ada.Task_Attributes);
```

4.a/2 **Discussion:** The Ravenscar profile is named for the location of the meeting that defined its initial version. The name is now in widespread use, so we stick with existing practice, rather than using a more descriptive name.

NOTES

5/2 41 {*AI95-00249-01*} The effect of the Max_Entry_Queue_Length => 1 restriction applies only to protected entry queues due to the accompanying restriction of Max_Task_Entries => 0.

Extensions to Ada 95

5.a/2 {*AI95-00296-01*} {*extensions to Ada 95*} The Ravenscar profile is new.

D.14 Execution Time

1/2 {*AI95-00307-01*} This clause describes a language-defined package to measure execution time.

Static Semantics

2/2 {*AI95-00307-01*} The following language-defined library package exists:

3/2
```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is
```

4/2
```
   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
   CPU_Time_Unit  : constant := implementation-defined-real-number;
   CPU_Tick : constant Time_Span;
```

5/2
```
   function Clock
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return CPU_Time;
```

6/2
```
   function "+"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   function "+"  (Left : Time_Span; Right : CPU_Time) return CPU_Time;
   function "-"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   function "-"  (Left : CPU_Time; Right : CPU_Time)  return Time_Span;
```

7/2
```
    function "<"  (Left, Right : CPU_Time) return Boolean;
    function "<=" (Left, Right : CPU_Time) return Boolean;
    function ">"  (Left, Right : CPU_Time) return Boolean;
    function ">=" (Left, Right : CPU_Time) return Boolean;
```

8/2
```
    procedure Split
      (T : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);
```

9/2
```
    function Time_Of (SC : Seconds_Count;
                      TS : Time_Span := Time_Span_Zero) return CPU_Time;
```

10/2
```
private
    ... -- not specified by the language
end Ada.Execution_Time;
```

11/2 {*AI95-00307-01*} {*execution time (of a task)*} {*CPU time (of a task)*} The *execution time* or CPU time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. It is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers and run-time services on behalf of the system.

11.a/2 **Discussion:** The implementation-defined properties above and of the values declared in the package are repeated in Documentation Requirements, so we don't mark them as implementation-defined.

12/2 {*AI95-00307-01*} The type CPU_Time represents the execution time of a task. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers.

13/2 {*AI95-00307-01*} The CPU_Time value I represents the half-open execution-time interval that starts with I*CPU_Time_Unit and is limited by (I+1)*CPU_Time_Unit, where CPU_Time_Unit is an implementation-defined real number. For each task, the execution time value is set to zero at the creation of the task.

13.a/2 **Ramification:** Since it is implementation-defined which task is charged execution time for system services, the execution time value may become non-zero even before the start of the activation of the task.

14/2 {*AI95-00307-01*} CPU_Time_First and CPU_Time_Last are the smallest and largest values of the CPU_Time type, respectively.

Dynamic Semantics

15/2 {*AI95-00307-01*} {*CPU clock tick*} CPU_Time_Unit is the smallest amount of execution time representable by the CPU_Time type; it is expressed in seconds. A *CPU clock tick* is an execution time interval during which the clock value (as observed by calling the Clock function) remains constant. CPU_Tick is the average length of such intervals.

16/2 {*AI95-00307-01*} The effects of the operators on CPU_Time and Time_Span are as for the operators defined for integer types.

17/2 {*AI95-00307-01*} The function Clock returns the current execution time of the task identified by T; Tasking_Error is raised if that task has terminated; Program_Error is raised if the value of T is Task_Identification.Null_Task_Id.

18/2 {*AI95-00307-01*} The effects of the Split and Time_Of operations are defined as follows, treating values of type CPU_Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split (T, SC, TS) is to set SC and TS to values such that T*CPU_Time_Unit = SC*1.0 + TS*CPU_Time_Unit, and 0.0 <= TS*CPU_Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the execution-time value T such that T*CPU_Time_Unit=SC*1.0 + TS*CPU_Time_Unit.

Erroneous Execution

19/2 {*AI95-00307-01*} {*erroneous execution (cause)* [partial]} For a call of Clock, if the task identified by T no longer exists, the execution of the program is erroneous.

Implementation Requirements

20/2 {*AI95-00307-01*} The range of CPU_Time values shall be sufficient to uniquely represent the range of execution times from the task start-up to 50 years of execution time later. CPU_Tick shall be no greater than 1 millisecond.

Documentation Requirements

21/2 {*AI95-00307-01*} The implementation shall document the values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick.

21.a/2 **Documentation Requirement:** The values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick of package Execution_Time.

22/2 {*AI95-00307-01*} The implementation shall document the properties of the underlying mechanism used to measure execution times, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

22.a/2 **Documentation Requirement:** The properties of the mechanism used to implement package Execution_Time.

Metrics

23/2 {*AI95-00307-01*} The implementation shall document the following metrics:

24/2
- An upper bound on the execution-time duration of a clock tick. This is a value D such that if t1 and t2 are any execution times of a given task such that t1 < t2 and $Clock_{t1}$ = $Clock_{t2}$ then t2 – t1 <= D.

25/2
- An upper bound on the size of a clock jump. A clock jump is the difference between two successive distinct values of an execution-time clock (as observed by calling the Clock function with the same Task_Id).

26/2
- An upper bound on the execution time of a call to the Clock function, in processor clock cycles.

27/2
- Upper bounds on the execution times of the operators of the type CPU_Time, in processor clock cycles.

27.a/2 **Documentation Requirement:** The metrics for execution time.

Implementation Permissions

28/2 {*AI95-00307-01*} Implementations targeted to machines with word size smaller than 32 bits need not support the full range and granularity of the CPU_Time type.

Implementation Advice

29/2 {*AI95-00307-01*} When appropriate, implementations should provide configuration mechanisms to change the value of CPU_Tick.

29.a/2 **Implementation Advice:** When appropriate, implementations should provide configuration mechanisms to change the value of Execution_Time.CPU_Tick.

Extensions to Ada 95

29.b/2 {*AI95-00307-01*} {*extensions to Ada 95*} The package Execution_Time is new.

D.14.1 Execution Time Timers

1/2 {*AI95-00307-01*} This clause describes a language-defined package that provides a facility for calling a handler when a task has used a defined amount of CPU time.

Static Semantics

2/2 {*AI95-00307-01*} The following language-defined library package exists:

```
3/2  with System;
     package Ada.Execution_Time.Timers is

4/2     type Timer (T : not null access constant
                            Ada.Task_Identification.Task_Id) is
           tagged limited private;

5/2     type Timer_Handler is
           access protected procedure (TM : in out Timer);

6/2     Min_Handler_Ceiling : constant System.Any_Priority :=
        implementation-defined;

7/2     procedure Set_Handler (TM      : in out Timer;
                               In_Time : in Time_Span;
                               Handler : in Timer_Handler);
        procedure Set_Handler (TM      : in out Timer;
                               At_Time : in CPU_Time;
                               Handler : in Timer_Handler);
        function Current_Handler (TM : Timer) return Timer_Handler;
        procedure Cancel_Handler (TM        : in out Timer;
                                  Cancelled :    out Boolean);

8/2     function Time_Remaining (TM : Timer) return Time_Span;

9/2     Timer_Resource_Error : exception;

10/2 private
        ... --  not specified by the language
     end Ada.Execution_Time.Timers;
```

11/2 {*AI95-00307-01*} The type Timer represents an execution-time event for a single task and is capable of detecting execution-time overruns. The access discriminant T identifies the task concerned. The type Timer needs finalization (see 7.6).

12/2 {*AI95-00307-01*} An object of type Timer is said to be *set* if it is associated with a non-null value of type Timer_Handler and *cleared* otherwise. All Timer objects are initially cleared. {*set (execution timer object)* [partial]} {*clear (execution timer object)* [partial]}

13/2 {*AI95-00307-01*} The type Timer_Handler identifies a protected procedure to be executed by the implementation when the timer expires. Such a protected procedure is called a *handler*. {*handler (execution timer)* [partial]}

> 13.a/2 **Discussion:** Type Timer is tagged. This makes it possible to share a handler between several events. In simple cases, 'Access can be used to compare the parameter with a specific timer object (this works because a tagged type is a by-reference type). In more complex cases, a type extension of type Timer can be declared; a double type conversion can be used to access the extension data. An example of how this can be done can be found for the similar type Timing_Event, see D.15.

Dynamic Semantics

14/2 {*AI95-00307-01*} When a Timer object is created, or upon the first call of a Set_Handler procedure with the timer as parameter, the resources required to operate an execution-time timer based on the associated execution-time clock are allocated and initialized. If this operation would exceed the available resources, Timer_Resource_Error is raised.

15/2 {*AI95-00307-01*} The procedures Set_Handler associate the handler Handler with the timer TM; if Handler is **null**, the timer is cleared, otherwise it is set. The first procedure Set_Handler loads the timer TM with an interval specified by the Time_Span parameter. In this mode, the timer TM *expires* when the execution time of the task identified by TM.T.**all** has increased by In_Time; if In_Time is less than or equal to zero, the timer expires immediately. The second procedure Set_Handler loads the timer TM with the absolute value specified by At_Time. In this mode, the timer TM expires when the execution time of the task identified by TM.T.**all** reaches At_Time; if the value of At_Time has already been reached when Set_Handler is called, the timer expires immediately.{*expires (execution timer)*}

15.a/2 **Implementation Note:** Since an access-to-constant can designate a variable, the Task_Id value designated by the discriminant of a Timer object can be changed after the object is created. Thus, an implementation cannot use the value of the Task_Id other than where this Standard specifies. For instance, the Task_Id should be read when the timer is set, but it should not be used when the timer expires (as it may designate a different task at that point.

16/2 {*AI95-00307-01*} A call of a procedure Set_Handler for a timer that is already set replaces the handler and the (absolute or relative) execution time; if Handler is not **null**, the timer remains set.

17/2 {*AI95-00307-01*} When a timer expires, the associated handler is executed, passing the timer as parameter. The initial action of the execution of the handler is to clear the event.

18/2 {*AI95-00307-01*} The function Current_Handler returns the handler associated with the timer TM if that timer is set; otherwise it returns **null**.

19/2 {*AI95-00307-01*} The procedure Cancel_Handler clears the timer if it is set. Cancelled is assigned True if the timer was set prior to it being cleared; otherwise it is assigned False.

20/2 {*AI95-00307-01*} The function Time_Remaining returns the execution time interval that remains until the timer TM would expire, if that timer is set; otherwise it returns Time_Span_Zero.

21/2 {*AI95-00307-01*} The constant Min_Handler_Ceiling is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked.

22/2 {*AI95-00307-01*} As part of the finalization of an object of type Timer, the timer is cleared.

23/2 {*AI95-00307-01*} For all the subprograms defined in this package, Tasking_Error is raised if the task identified by TM.T.**all** has terminated, and Program_Error is raised if the value of TM.T.**all** is Task_Identification.Null_Task_Id.

24/2 {*AI95-00307-01*} An exception propagated from a handler invoked as part of the expiration of a timer has no effect.

Erroneous Execution

25/2 {*AI95-00307-01*} {*erroneous execution (cause)* [partial]} For a call of any of the subprograms defined in this package, if the task identified by TM.T.**all** no longer exists, the execution of the program is erroneous.

Implementation Requirements

26/2 {*AI95-00307-01*} For a given Timer object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timer object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

26.a/2 **Reason:** This prevents various race conditions. In particular it ensures that if an event occurs when Set_Handler is changing the handler then either the new or old handler is executed in response to the appropriate event. It is never possible for a new handler to be executed in response to an old event

27/2 {*AI95-00307-01*} When an object of type Timer is finalized, the system resources used by the timer shall be deallocated.

Implementation Permissions

28/2 {*AI95-00307-01*} Implementations may limit the number of timers that can be defined for each task. If this limit is exceeded then Timer_Resource_Error is raised.

NOTES

29/2 42 {*AI95-00307-01*} A Timer_Handler can be associated with several Timer objects.

Extensions to Ada 95

{*AI95-00307-01*} {*extensions to Ada 95*} The package Execution_Time.Timers is new. 29.a/2

D.14.2 Group Execution Time Budgets

{*AI95-00354-01*} This clause describes a language-defined package to assign execution time budgets to groups of tasks. 1/2

Static Semantics

{*AI95-00354-01*} The following language-defined library package exists: 2/2

```
with System;                                                          3/2
package Ada.Execution_Time.Group_Budgets is

  type Group_Budget is tagged limited private;                        4/2

  type Group_Budget_Handler is access                                 5/2
       protected procedure (GB : in out Group_Budget);

  type Task_Array is array (Positive range <>) of                     6/2
                                  Ada.Task_Identification.Task_Id;

  Min_Handler_Ceiling : constant System.Any_Priority :=               7/2
    implementation-defined;

  procedure Add_Task (GB : in out Group_Budget;                       8/2
                      T  : in Ada.Task_Identification.Task_Id);
  procedure Remove_Task (GB: in out Group_Budget;
                         T  : in Ada.Task_Identification.Task_Id);
  function Is_Member (GB : Group_Budget;
                      T : Ada.Task_Identification.Task_Id) return Boolean;
  function Is_A_Group_Member
     (T : Ada.Task_Identification.Task_Id) return Boolean;
  function Members (GB : Group_Budget) return Task_Array;

  procedure Replenish (GB : in out Group_Budget; To : in Time_Span);  9/2
  procedure Add (GB : in out Group_Budget; Interval : in Time_Span);
  function Budget_Has_Expired (GB : Group_Budget) return Boolean;
  function Budget_Remaining (GB : Group_Budget) return Time_Span;

  procedure Set_Handler (GB      : in out Group_Budget;               10/2
                         Handler : in Group_Budget_Handler);
  function Current_Handler (GB : Group_Budget)
     return Group_Budget_Handler;
  procedure Cancel_Handler (GB        : in out Group_Budget;
                            Cancelled : out Boolean);

  Group_Budget_Error : exception;                                     11/2

private                                                               12/2
    --  not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

{*AI95-00354-01*} The type Group_Budget represents an execution time budget to be used by a group of tasks. The type Group_Budget needs finalization (see 7.6). A task can belong to at most one group. Tasks of any priority can be added to a group. 13/2

{*AI95-00354-01*} An object of type Group_Budget has an associated nonnegative value of type Time_Span known as its *budget*, which is initially Time_Span_Zero. The type Group_Budget_Handler identifies a protected procedure to be executed by the implementation when the budget is *exhausted*, that is, reaches zero. Such a protected procedure is called a *handler*.{*budget*} {*exhaust (a budget)*} {*handler (group budget)* [partial]} 14/2

{*AI95-00354-01*} An object of type Group_Budget also includes a handler, which is a value of type Group_Budget_Handler. The handler of the object is said to be *set* if it is not null and *cleared* otherwise. 15/2

The handler of all Group_Budget objects is initially cleared. {*set (group budget object)* [partial]} {*clear (group budget object)* [partial]}

15.a/2 **Discussion:** Type Group_Budget is tagged. This makes it possible to share a handler between several events. In simple cases, 'Access can be used to compare the parameter with a specific group budget object (this works because a tagged type is a by-reference type). In more complex cases, a type extension of type Group_Budget can be declared; a double type conversion can be used to access the extension data. An example of how this can be done can be found for the similar type Timing_Event, see D.15.

Dynamic Semantics

16/2 {*AI95-00354-01*} The procedure Add_Task adds the task identified by T to the group GB; if that task is already a member of some other group, Group_Budget_Error is raised.

17/2 {*AI95-00354-01*} The procedure Remove_Task removes the task identified by T from the group GB; if that task is not a member of the group GB, Group_Budget_Error is raised. After successful execution of this procedure, the task is no longer a member of any group.

18/2 {*AI95-00354-01*} The function Is_Member returns True if the task identified by T is a member of the group GB; otherwise it return False.

19/2 {*AI95-00354-01*} The function Is_A_Group_Member returns True if the task identified by T is a member of some group; otherwise it returns False.

20/2 {*AI95-00354-01*} The function Members returns an array of values of type Task_Identification.Task_Id identifying the members of the group GB. The order of the components of the array is unspecified.

21/2 {*AI95-00354-01*} The procedure Replenish loads the group budget GB with To as the Time_Span value. The exception Group_Budget_Error is raised if the Time_Span value To is non-positive. Any execution of any member of the group of tasks results in the budget counting down, unless exhausted. When the budget becomes exhausted (reaches Time_Span_Zero), the associated handler is executed if the handler of group budget GB is set. Nevertheless, the tasks continue to execute.

22/2 {*AI95-00354-01*} The procedure Add modifies the budget of the group GB. A positive value for Interval increases the budget. A negative value for Interval reduces the budget, but never below Time_Span_Zero. A zero value for Interval has no effect. A call of procedure Add that results in the value of the budget going to Time_Span_Zero causes the associated handler to be executed if the handler of the group budget GB is set.

23/2 {*AI95-00354-01*} The function Budget_Has_Expired returns True if the budget of group GB is exhausted (equal to Time_Span_Zero); otherwise it returns False.

24/2 {*AI95-00354-01*} The function Budget_Remaining returns the remaining budget for the group GB. If the budget is exhausted it returns Time_Span_Zero. This is the minimum value for a budget.

25/2 {*AI95-00354-01*} The procedure Set_Handler associates the handler Handler with the Group_Budget GB; if Handler is **null**, the handler of Group_Budget is cleared, otherwise it is set.

26/2 {*AI95-00354-01*} A call of Set_Handler for a Group_Budget that already has a handler set replaces the handler; if Handler is not **null**, the handler for Group_Budget remains set.

27/2 {*AI95-00354-01*} The function Current_Handler returns the handler associated with the group budget GB if the handler for that group budget is set; otherwise it returns **null**.

28/2 {*AI95-00354-01*} The procedure Cancel_Handler clears the handler for the group budget if it is set. Cancelled is assigned True if the handler for the group budget was set prior to it being cleared; otherwise it is assigned False.

{*AI95-00354-01*} The constant Min_Handler_Ceiling is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked. 29/2

{*AI95-00354-01*} The precision of the accounting of task execution time to a Group_Budget is the same as that defined for execution-time clocks from the parent package. 30/2

{*AI95-00354-01*} As part of the finalization of an object of type Group_Budget all member tasks are removed from the group identified by that object. 31/2

{*AI95-00354-01*} If a task is a member of a Group_Budget when it terminates then as part of the finalization of the task it is removed from the group. 32/2

{*AI95-00354-01*} For all the operations defined in this package, Tasking_Error is raised if the task identified by T has terminated, and Program_Error is raised if the value of T is Task_Identification.Null_Task_Id. 33/2

{*AI95-00354-01*} An exception propagated from a handler invoked when the budget of a group of tasks becomes exhausted has no effect. 34/2

Erroneous Execution

{*AI95-00354-01*} {*erroneous execution (cause)* [partial]} For a call of any of the subprograms defined in this package, if the task identified by T no longer exists, the execution of the program is erroneous. 35/2

Implementation Requirements

{*AI95-00354-01*} For a given Group_Budget object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Group_Budget object. The replacement of a handler, by a call of Set_Handler, shall be performed atomically with respect to the execution of the handler. 36/2

Reason: This prevents various race conditions. In particular it ensures that if the budget is exhausted when Set_Handler is changing the handler then either the new or old handler is executed and the exhausting event is not lost. 36.a/2

NOTES

43 {*AI95-00354-01*} Clearing or setting of the handler of a group budget does not change the current value of the budget. Exhaustion or loading of a budget does not change whether the handler of the group budget is set or cleared. 37/2

44 {*AI95-00354-01*} A Group_Budget_Handler can be associated with several Group_Budget objects. 38/2

Extensions to Ada 95

{*AI95-00354-01*} {*extensions to Ada 95*} The package Execution_Time.Group_Budgets is new. 38.a/2

D.15 Timing Events

{*AI95-00297-01*} This clause describes a language-defined package to allow user-defined protected procedures to be executed at a specified time without the need for a task or a delay statement. 1/2

Static Semantics

{*AI95-00297-01*} The following language-defined library package exists: 2/2

```
package Ada.Real_Time.Timing_Events is                              3/2

  type Timing_Event is tagged limited private;                      4/2
  type Timing_Event_Handler
       is access protected procedure (Event : in out Timing_Event);
```

5/2
```
      procedure Set_Handler (Event   : in out Timing_Event;
                             At_Time : in Time;
                             Handler : in Timing_Event_Handler);
      procedure Set_Handler (Event   : in out Timing_Event;
                             In_Time : in Time_Span;
                             Handler : in Timing_Event_Handler);
      function Current_Handler (Event : Timing_Event)
           return Timing_Event_Handler;
      procedure Cancel_Handler (Event     : in out Timing_Event;
                                Cancelled : out Boolean);
```

6/2
```
      function Time_Of_Event (Event : Timing_Event) return Time;
```

7/2
```
   private
      ... -- not specified by the language
   end Ada.Real_Time.Timing_Events;
```

8/2 {*AI95-00297-01*} The type Timing_Event represents a time in the future when an event is to occur. The type Timing_Event needs finalization (see 7.6).

9/2 {*AI95-00297-01*} An object of type Timing_Event is said to be *set* if it is associated with a non-null value of type Timing_Event_Handler and *cleared* otherwise. All Timing_Event objects are initially cleared. {*set (timing event object)* [partial]} {*clear (timing event object)* [partial]}

10/2 {*AI95-00297-01*} The type Timing_Event_Handler identifies a protected procedure to be executed by the implementation when the timing event occurs. Such a protected procedure is called a *handler*. {*handler (timing event)* [partial]}

10.a/2 **Discussion:** Type Timing_Event is tagged. This makes it possible to share a handler between several events. In simple cases, 'Access can be used to compare the parameter with a specific timing event object (this works because a tagged type is a by-reference type). In more complex cases, a type extension of type Timing_Event can be declared; a double type conversion can be used to access the extension data. For example:

10.b/2
```
type Toaster_Timing_Event is new Timing_Event with record
   Slot : Natural;
end record;
```

10.c/2
```
...
```

10.d/2
```
protected body Toaster is
```

10.e/2
```
   procedure Timer(Event : in out Timing_Event) is
   begin
      Pop_Up_Toast (Toaster_Timing_Event(Timing_Event'Class(Event)).Slot);
   end Timer;
```

10.f/2
```
   ...
end Toaster;
```

10.g/2 The extra conversion to the class-wide type is necessary to make the conversions legal. While this usage is clearly ugly, we think that the need for this sort of usage will be rare, so we can live with it. It's certainly better than having no way to associate data with an event.

Dynamic Semantics

11/2 {*AI95-00297-01*} The procedures Set_Handler associate the handler Handler with the event Event; if Handler is **null**, the event is cleared, otherwise it is set. The first procedure Set_Handler sets the execution time for the event to be At_Time. The second procedure Set_Handler sets the execution time for the event to be Real_Time.Clock + In_Time.

12/2 {*AI95-00297-01*} A call of a procedure Set_Handler for an event that is already set replaces the handler and the time of execution; if Handler is not **null**, the event remains set.

13/2 {*AI95-00297-01*} As soon as possible after the time set for the event, the handler is executed, passing the event as parameter. The handler is only executed if the timing event is in the set state at the time of execution. The initial action of the execution of the handler is to clear the event.

Reason: The second sentence of this paragraph is because of a potential race condition. The time might expire and yet before the handler is executed, some task could call Cancel_Handler (or equivalently call Set_Handler with a **null** parameter) and thus clear the handler. 13.a/2

{*AI95-00297-01*} If the Ceiling_Locking policy (see D.3) is in effect when a procedure Set_Handler is called, a check is made that the ceiling priority of Handler.**all** is Interrupt_Priority'Last. If the check fails, Program_Error is raised. 14/2

{*AI95-00297-01*} If a procedure Set_Handler is called with zero or negative In_Time or with At_Time indicating a time in the past then the handler is executed immediately by the task executing the call of Set_Handler. The timing event Event is cleared. 15/2

{*AI95-00297-01*} The function Current_Handler returns the handler associated with the event Event if that event is set; otherwise it returns **null**. 16/2

{*AI95-00297-01*} The procedure Cancel_Handler clears the event if it is set. Cancelled is assigned True if the event was set prior to it being cleared; otherwise it is assigned False. 17/2

{*AI95-00297-01*} The function Time_Of_Event returns the time of the event if the event is set; otherwise it returns Real_Time.Time_First. 18/2

{*AI95-00297-01*} As part of the finalization of an object of type Timing_Event, the Timing_Event is cleared. 19/2

Implementation Note: This is the only finalization defined by the language that has a visible effect; but an implementation may have other finalization that it needs to perform. Implementations need to ensure that the event is cleared before anything else is finalized that would prevent a set event from being triggered. 19.a/2

{*AI95-00297-01*} If several timing events are set for the same time, they are executed in FIFO order of being set. 20/2

{*AI95-00297-01*} An exception propagated from a handler invoked by a timing event has no effect. 21/2

Implementation Requirements

{*AI95-00297-01*} For a given Timing_Event object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timing_Event object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler. 22/2

Reason: This prevents various race conditions. In particular it ensures that if an event occurs when Set_Handler is changing the handler then either the new or old handler is executed in response to the appropriate event. It is never possible for a new handler to be executed in response to an old event. 22.a/2

Metrics

{*AI95-00297-01*} The implementation shall document the following metric: 23/2

- An upper bound on the lateness of the execution of a handler. That is, the maximum time between when a handler is actually executed and the time specified when the event was set. 24/2

Documentation Requirement: The metrics for timing events. 24.a/2

Implementation Advice

{*AI95-00297-01*} The protected handler procedure should be executed directly by the real-time clock interrupt mechanism. 25/2

Implementation Advice: For a timing event, the handler should be executed directly by the real-time clock interrupt mechanism. 25.a/2

NOTES

45 {*AI95-00297-01*} Since a call of Set_Handler is not a potentially blocking operation, it can be called from within a handler. 26/2

27/2 46 {*AI95-00297-01*} A Timing_Event_Handler can be associated with several Timing_Event objects.

Extensions to Ada 95

27.a/2 {*AI95-00297-01*} {*extensions to Ada 95*} The package Real_Time.Timing_Events is new.

Annex E
(normative)
Distributed Systems

[This Annex defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program.] 1

Extensions to Ada 83

{*extensions to Ada 83*} This Annex is new to Ada 95. 1.a

Post-Compilation Rules

{*processing node*} {*storage node*} {*distributed system*} A *distributed system* is an interconnection of one or more *processing nodes* (a system resource that has both computational and storage capabilities), and zero or more *storage nodes* (a system resource that has only storage capabilities, with the storage addressable by one or more processing nodes). 2

{*distributed program*} A *distributed program* comprises one or more partitions that execute independently (except when they communicate) in a distributed system. 3

{*configuration (of the partitions of a program)*} The process of mapping the partitions of a program to the nodes in a distributed system is called *configuring the partitions of the program*. 4

Implementation Requirements

The implementation shall provide means for explicitly assigning library units to a partition and for the configuring and execution of a program consisting of multiple partitions on a distributed system; the means are implementation defined. 5

Implementation defined: The means for creating and executing distributed programs. 5.a

Implementation Permissions

An implementation may require that the set of processing nodes of a distributed system be homogeneous. 6

NOTES
1 The partitions comprising a program may be executed on differently configured distributed systems or on a non-distributed system without requiring recompilation. A distributed program may be partitioned differently from the same set of library units without recompilation. The resulting execution is semantically equivalent. 7

2 A distributed program retains the same type safety as the equivalent single partition program. 8

E.1 Partitions

[The partitions of a distributed program are classified as either active or passive.] 1

Post-Compilation Rules

{*active partition*} {*passive partition*} An *active partition* is a partition as defined in 10.2. A *passive partition* is a partition that has no thread of control of its own, whose library units are all preelaborated, and whose data and subprograms are accessible to one or more active partitions. 2

Discussion: In most situations, a passive partition does not execute, and does not have a “real” environment task. Any execution involved in its elaboration and initialization occurs before it comes into existence in a distributed program (like most preelaborated entities). Likewise, there is no concrete meaning to passive partition termination. 2.a

A passive partition shall include only library_items that either are declared pure or are shared passive (see 10.2.1 and E.2.1). 3

4 An active partition shall be configured on a processing node. A passive partition shall be configured either on a storage node or on a processing node.

5 The configuration of the partitions of a program onto a distributed system shall be consistent with the possibility for data references or calls between the partitions implied by their semantic dependences. {*remote access*} Any reference to data or call of a subprogram across partitions is called a *remote access*.

5.a **Discussion:** For example, an active partition that includes a unit with a semantic dependence on the declaration of another RCI package of some other active partition has to be connected to that other partition by some sort of a message passing mechanism.

5.b A passive partition that is accessible to an active partition should have its storage addressable to the processor(s) of the active partition. The processor(s) should be able to read and write from/to that storage, as well as to perform “read-modify-write” operations (in order to support entry-less protected objects).

Dynamic Semantics

6 {*elaboration (partition)*} A library_item is elaborated as part of the elaboration of each partition that includes it. If a normal library unit (see E.2) has state, then a separate copy of the state exists in each active partition that elaborates it. [The state evolves independently in each such partition.]

6.a **Ramification:** Normal library units cannot be included in passive partitions.

7 {*termination (of a partition)*} {*abort (of a partition)*} {*inaccessible partition*} {*accessible partition*} [An active partition *terminates* when its environment task terminates.] A partition becomes *inaccessible* if it terminates or if it is *aborted*. An active partition is aborted when its environment task is aborted. In addition, if a partition fails during its elaboration, it becomes inaccessible to other partitions. Other implementation-defined events can also result in a partition becoming inaccessible.

7.a **Implementation defined:** Any events that can result in a partition becoming inaccessible.

8/1 For a prefix D that denotes a library-level declaration, excepting a declaration of or within a declared-pure library unit, the following attribute is defined:

9 D'Partition_Id

Denotes a value of the type *universal_integer* that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see E.2.3) the given partition is the one where the body of D was elaborated.

Bounded (Run-Time) Errors

10 {*bounded error (cause)* [partial]} It is a bounded error for there to be cyclic elaboration dependences between the active partitions of a single distributed program. {*Program_Error (raised by failure of run-time check)*} The possible effects, in each of the partitions involved, are deadlock during elaboration, or the raising of Communication_Error or Program_Error.

Implementation Permissions

11 An implementation may allow multiple active or passive partitions to be configured on a single processing node, and multiple passive partitions to be configured on a single storage node. In these cases, the scheduling policies, treatment of priorities, and management of shared resources between these partitions are implementation defined.

11.a **Implementation defined:** The scheduling policies, treatment of priorities, and management of shared resources between partitions in certain cases.

12 An implementation may allow separate copies of an active partition to be configured on different processing nodes, and to provide appropriate interactions between the copies to present a consistent state of the partition to other active partitions.

12.a **Ramification:** The language does not specify the nature of these interactions, nor the actual level of consistency preserved.

13 In an implementation, the partitions of a distributed program need not be loaded and elaborated all at the same time; they may be loaded and elaborated one at a time over an extended period of time. An implementation may provide facilities to abort and reload a partition during the execution of a distributed program.

14 An implementation may allow the state of some of the partitions of a distributed program to persist while other partitions of the program terminate and are later reinvoked.

NOTES

15 3 Library units are grouped into partitions after compile time, but before run time. At compile time, only the relevant library unit properties are identified using categorization pragmas.

16 4 The value returned by the Partition_Id attribute can be used as a parameter to implementation-provided subprograms in order to query information about the partition.

Wording Changes from Ada 95

16.a/2 {*AI95-00226-01*} Corrected wording so that a partition that has an elaboration problem will either deadlock or raise an exception. While an Ada 95 implementation could allow some partitions to continue to execute, they could be accessing unelaborated data, which is very bad (and erroneous in a practical sense). Therefore, this isn't listed as an inconsistency.

E.2 Categorization of Library Units

1 [Library units can be categorized according to the role they play in a distributed program. Certain restrictions are associated with each category to ensure that the semantics of a distributed program remain close to the semantics for a nondistributed program.]

2 {*categorization pragma* [distributed]} {*pragma, categorization* [distributed]} {*library unit pragma (categorization pragmas)* [partial]} {*pragma, library unit (categorization pragmas)* [partial]} {*categorized library unit*} A *categorization pragma* is a library unit pragma (see 10.1.5) that restricts the declarations, child units, or semantic dependences of the library unit to which it applies. A *categorized library unit* is a library unit to which a categorization pragma applies.

3 The pragmas Shared_Passive, Remote_Types, and Remote_Call_Interface are categorization pragmas. In addition, for the purposes of this Annex, the pragma Pure (see 10.2.1) is considered a categorization pragma.

4/1 {*8652/0078*} {*AI95-00048-01*} {*shared passive library unit*} A library package or generic library package is called a *shared passive* library unit if a Shared_Passive pragma applies to it. {*remote types library unit*} A library package or generic library package is called a *remote types* library unit if a Remote_Types pragma applies to it. {*remote call interface*} A library unit is called a *remote call interface* if a Remote_Call_Interface pragma applies to it. {*normal library unit*} A *normal library unit* is one to which no categorization pragma applies.

4.a.1/1 **Ramification:** {*8652/0078*} {*AI95-00048-01*} A library subprogram can be a remote call interface, but it cannot be a remote types or shared passive library unit.

5 [The various categories of library units and the associated restrictions are described in this clause and its subclauses. The categories are related hierarchically in that the library units of one category can depend semantically only on library units of that category or an earlier one, except that the body of a remote types or remote call interface library unit is unrestricted.

6 The overall hierarchy (including declared pure) is as follows:

7 Declared Pure
Can depend only on other declared pure library units;

8 Shared Passive
Can depend only on other shared passive or declared pure library units;

9 Remote Types
The declaration of the library unit can depend only on other remote types library units, or one of the above; the body of the library unit is unrestricted;

10 Remote Call Interface
The declaration of the library unit can depend only on other remote call interfaces, or one of the above; the body of the library unit is unrestricted;

11 Normal Unrestricted.

12 Declared pure and shared passive library units are preelaborated. The declaration of a remote types or remote call interface library unit is required to be preelaborable.]

Implementation Requirements

13/1 *This paragraph was deleted.*{*8652/0079*} {*AI95-00208-01*}

Implementation Permissions

14 Implementations are allowed to define other categorization pragmas.

Wording Changes from Ada 95

14.a/2 {*8652/0078*} {*AI95-00048-01*} **Corrigendum:** Clarified that a library subprogram can be a remote call interface unit.

14.b/2 {*8652/0079*} {*AI95-00208-01*} **Corrigendum:** Removed the requirement that types be represented the same in all partitions, because it prevents the definition of heterogeneous distributed systems and goes much further than required.

E.2.1 Shared Passive Library Units

1 [A shared passive library unit is used for managing global data shared between active partitions. The restrictions on shared passive library units prevent the data or tasks of one active partition from being accessible to another active partition through references implicit in objects declared in the shared passive library unit.]

Language Design Principles

1.a The restrictions governing a shared passive library unit are designed to ensure that objects and subprograms declared in the package can be used safely from multiple active partitions, even though the active partitions live in different address spaces, and have separate run-time systems.

Syntax

2 {*categorization pragma (Shared_Passive)* [partial]} {*pragma, categorization (Shared_Passive)* [partial]} The form of a pragma Shared_Passive is as follows:

3 **pragma** Shared_Passive[(*library_unit*_name)];

Legality Rules

4 {*shared passive library unit*} A *shared passive library unit* is a library unit to which a Shared_Passive pragma applies. The following restrictions apply to such a library unit:

5 - [it shall be preelaborable (see 10.2.1);]

5.a **Ramification:** It cannot contain library-level declarations of protected objects with entries, nor of task objects. Task objects are disallowed because passive partitions don't have any threads of control of their own, nor any run-time system of their own. Protected objects with entries are disallowed because an entry queue contains references to calling tasks, and that would require in effect a pointer from a passive partition back to a task in some active partition.

6 - it shall depend semantically only upon declared pure or shared passive library units;

Reason: Shared passive packages cannot depend semantically upon remote types packages because the values of an access type declared in a remote types package refer to the local heap of the active partition including the remote types package. 6.a

- {*8652/0080*} {*AI95-00003-01*} it shall not contain a library-level declaration of an access type that designates a class-wide type, task type, or protected type with entry_declarations. 7/1

Reason: These kinds of access types are disallowed because the object designated by an access value of such a type could contain an implicit reference back to the active partition on whose behalf the designated object was created. 7.a

{*accessibility (from shared passive library units)* [partial]} {*notwithstanding*} Notwithstanding the definition of accessibility given in 3.10.2, the declaration of a library unit P1 is not accessible from within the declarative region of a shared passive library unit P2, unless the shared passive library unit P2 depends semantically on P1. 8

Discussion: We considered a more complex rule, but dropped it. This is the simplest rule that recognizes that a shared passive package may outlive some other library package, unless it depends semantically on that package. In a nondistributed program, all library packages are presumed to have the same lifetime. 8.a

Implementations may define additional pragmas that force two library packages to be in the same partition, or to have the same lifetime, which would allow this rule to be relaxed in the presence of such pragmas. 8.b

Static Semantics

{*preelaborated* [partial]} A shared passive library unit is preelaborated. 9

Post-Compilation Rules

A shared passive library unit shall be assigned to at most one partition within a given program. 10

{*compilation units needed (shared passive library unit)* [partial]} {*needed (shared passive library unit)* [partial]} {*notwithstanding*} Notwithstanding the rule given in 10.2, a compilation unit in a given partition does not *need* (in the sense of 10.2) the shared passive library units on which it depends semantically to be included in that same partition; they will typically reside in separate passive partitions. 11

Wording Changes from Ada 95

{*8652/0080*} {*AI95-00003-01*} **Corrigendum:** Corrected the wording to allow access types in blocks in shared passive generic packages. 11.a/2

E.2.2 Remote Types Library Units

[A remote types library unit supports the definition of types intended for use in communication between active partitions.] 1

Language Design Principles

The restrictions governing a remote types package are similar to those for a declared pure package. However, the restrictions are relaxed deliberately to allow such a package to contain declarations that violate the stateless property of pure packages, though it is presumed that any state-dependent properties are essentially invisible outside the package. 1.a

Syntax

{*categorization pragma (Remote_Types)* [partial]} {*pragma, categorization (Remote_Types)* [partial]} The form of a pragma Remote_Types is as follows: 2

pragma Remote_Types[(*library_unit_*name)]; 3

Legality Rules

{*remote types library unit*} A *remote types library unit* is a library unit to which the pragma Remote_Types applies. The following restrictions apply to the declaration of such a library unit: 4

- [it shall be preelaborable;] 5

6 • it shall depend semantically only on declared pure, shared passive, or other remote types library units;

7 • it shall not contain the declaration of any variable within the visible part of the library unit;

7.a **Reason:** This is essentially a "methodological" restriction. A separate copy of a remote types package is included in each partition that references it, just like a normal package. Nevertheless, a remote types package is thought of as an "essentially pure" package for defining types to be used for interpartition communication, and it could be misleading to declare visible objects when no remote data access is actually being provided.

8/2 • {*AI95-00240-01*} {*AI95-00366-01*} the full view of each type declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see 13.13.2).

8.a **Reason:** This is to prevent the use of the predefined Read and Write attributes of an access type as part of the Read and Write attributes of a visible type.

8.b/2 **Ramification:** {*AI95-00366-01*} Types that do not have available stream attributes are excluded from this rule; that means that attributes do not need to be specified for most limited types. It is only necessary to specify attributes for nonlimited types that have a part that is of any access type, and for extensions of limited types with available stream attributes where the record_extension_part includes a subcomponent of an access type, where the access type does not have specified attributes.

9/1 {*8652/0082*} {*AI95-00164-01*} {*remote access type*} An access type declared in the visible part of a remote types or remote call interface library unit is called a *remote access type*. {*remote access-to-subprogram type*} {*remote access-to-class-wide type*} Such a type shall be:

9.1/1 • {*8652/0082*} {*AI95-00164-01*} an access-to-subprogram type, or

9.2/1 • {*8652/0082*} {*AI95-00164-01*} a general access type that designates a class-wide limited private type or a class-wide private type extension all of whose ancestors are either private type extensions or limited private types.

9.3/1 {*8652/0081*} {*AI95-00004-01*} A type that is derived from a remote access type is also a remote access type.

10 The following restrictions apply to the use of a remote access-to-subprogram type:

11/2 • {*AI95-00431-01*} A value of a remote access-to-subprogram type shall be converted only to or from another (subtype-conformant) remote access-to-subprogram type;

12 • The prefix of an Access attribute_reference that yields a value of a remote access-to-subprogram type shall statically denote a (subtype-conformant) remote subprogram.

13 The following restrictions apply to the use of a remote access-to-class-wide type:

14/2 • {*8652/0083*} {*AI95-00047-01*} {*AI95-00240-01*} {*AI95-00366-01*} The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall support external streaming (see 13.13.2);

15 • A value of a remote access-to-class-wide type shall be explicitly converted only to another remote access-to-class-wide type;

16/1 • A value of a remote access-to-class-wide type shall be dereferenced (or implicitly converted to an anonymous access type) only as part of a dispatching call where the value designates a controlling operand of the call (see E.4, "Remote Subprogram Calls").

17/2 • {*AI95-00366-01*} The Storage_Pool attribute is not defined for a remote access-to-class-wide type; the expected type for an allocator shall not be a remote access-to-class-wide type. A remote access-to-class-wide type shall not be an actual parameter for a generic formal access type. The Storage_Size attribute of a remote access-to-class-wide type yields 0; it is not allowed in an attribute_definition_clause.

17.a/2 **Reason:** All three of these restrictions are because there is no storage pool associated with a remote access-to-class-wide type. The Storage_Size is defined to be 0 so that there is no conflict with the rules for pure units.

NOTES

18 5 A remote types library unit need not be pure, and the types it defines may include levels of indirection implemented by using access types. User-specified Read and Write attributes (see 13.13.2) provide for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling any levels of indirection.

Incompatibilities With Ada 95

18.a/2 {*AI95-00240-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** The wording was changed from "user-specified" to "available" attributes. (This was then further changed, see below.) This means that an access type with the attributes specified in the private part would originally have been sufficient to allow the access type to be used in a remote type, but that is no longer allowed. Similarly, the attributes of a remote type that has access components have to be specified in the visible part. These changes were made so that the rules were consistent with the rules introduced for the Corrigendum for stream attributes; moreover, legality should not depend on the contents of the private part.

Extensions to Ada 95

18.b/2 {*AI95-00366-01*} {*extensions to Ada 95*} Remote types that cannot be streamed (that is, have no available stream attributes) do not require the specification of stream attributes. This is necessary so that most extensions of Limited_Controlled do not need stream attributes defined (otherwise there would be a signficant incompatibility, as Limited_Controlled would need stream attributes, and then all extensions of it also would need stream attributes).

Wording Changes from Ada 95

18.c/2 {*8652/0081*} {*AI95-00004-01*} **Corrigendum:** Added missing wording so that a type derived from a remote access type is also a remote access type.

18.d/2 {*8652/0083*} {*AI95-00047-01*} **Corrigendum:** Clarified that user-defined Read and Write attributes are required for the primitive subprograms corresponding to a remote access-to-class-wide type.

18.e/2 {*8652/0082*} {*AI95-00164-01*} **Corrigendum:** Added missing wording so that a remote access type can designate an appropriate private extension.

18.f/2 {*AI95-00366-01*} Changed the wording to use the newly defined term *type that supports external streaming*, so that various issues with access types in pure units and implicitly declared attributes for type extensions are properly handled.

18.g/2 {*AI95-00366-01*} Defined Storage_Size to be 0 for remote access-to-class-wide types, rather than having it undefined. This eliminates issues with pure units requiring a defined storage size.

18.h/2 {*AI95-00431-01*} Corrected the wording so that a value of a local access-to-subprogram type cannot be converted to a remote access-to-subprogram type, as intended (and required by the ACATS).

E.2.3 Remote Call Interface Library Units

1 [A remote call interface library unit can be used as an interface for remote procedure calls (RPCs) (or remote function calls) between active partitions.]

Language Design Principles

1.a The restrictions governing a remote call interface library unit are intended to ensure that the values of the actual parameters in a remote call can be meaningfully sent between two active partitions.

Syntax

2 {*categorization pragma (Remote_Call_Interface)* [partial]} {*pragma, categorization (Remote_Call_Interface)* [partial]} The form of a pragma Remote_Call_Interface is as follows:

3 **pragma** Remote_Call_Interface[(*library_unit*_name)];

4 The form of a pragma All_Calls_Remote is as follows:

5 **pragma** All_Calls_Remote[(*library_unit*_name)];

6 {*library unit pragma (All_Calls_Remote)* [partial]} {*pragma, library unit (All_Calls_Remote)* [partial]} A pragma All_Calls_Remote is a library unit pragma.

Legality Rules

7/1 {*8652/0078*} {*AI95-00048-01*} {*remote call interface*} {*RCI (library unit)*} {*RCI (package)*} {*RCI (generic)*} {*remote subprogram*} A *remote call interface (RCI)* is a library unit to which the pragma Remote_Call_Interface applies. A subprogram declared in the visible part of such a library unit, or declared by such a library unit, is called a *remote subprogram*.

8 The declaration of an RCI library unit shall be preelaborable (see 10.2.1), and shall depend semantically only upon declared pure, shared passive, remote types, or other remote call interface library units.

9/1 {*8652/0078*} {*AI95-00048-01*} In addition, the following restrictions apply to an RCI library unit:

10/1 • {*8652/0078*} {*AI95-00048-01*} its visible part shall not contain the declaration of a variable;

10.a/1 **Reason:** {*8652/0078*} {*AI95-00048-01*} Remote call interface units do not provide remote data access. A shared passive package has to be used for that.

11/1 • {*8652/0078*} {*AI95-00048-01*} its visible part shall not contain the declaration of a limited type;

11.a/2 **Reason:** {*AI95-00240-01*} {*AI95-00366-01*} We disallow the declaration of task and protected types, since calling an entry or a protected subprogram implicitly passes an object of a limited type (the target task or protected object). We disallow other limited types since we require that such types have available Read and Write attributes, but we certainly don't want the Read and Write attributes themselves to involve remote calls (thereby defeating their purpose of marshalling the value for remote calls).

12/1 • {*8652/0078*} {*AI95-00048-01*} its visible part shall not contain a nested generic_declaration;

12.a **Reason:** This is disallowed because the body of the nested generic would presumably have access to data inside the body of the RCI package, and if instantiated in a different partition, remote data access might result, which is not supported.

13/1 • {*8652/0078*} {*AI95-00048-01*} it shall not be, nor shall its visible part contain, the declaration of a subprogram to which a pragma Inline applies;

14/2 • {*8652/0078*} {*AI95-00048-01*} {*AI95-00240-01*} {*AI95-00366-01*} it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has [an access parameter or] a parameter of a type that does not support external streaming (see 13.13.2);

15 • any public child of the library unit shall be a remote call interface library unit.

15.a **Reason:** No restrictions apply to the private part of an RCI package, and since a public child can "see" the private part of its parent, such a child must itself have a Remote_Call_Interface pragma, and be assigned to the same partition (see below).

15.b **Discussion:** We considered making the public child of an RCI package implicitly RCI, but it seemed better to require an explicit pragma to avoid any confusion.

15.c Note that there is no need for a private child to be an RCI package, since it can only be seen from the body of its parent or its siblings, all of which are required to be in the same active partition.

16 If a pragma All_Calls_Remote applies to a library unit, the library unit shall be a remote call interface.

Post-Compilation Rules

17 A remote call interface library unit shall be assigned to at most one partition of a given program. A remote call interface library unit whose parent is also an RCI library unit shall be assigned only to the same partition as its parent.

17.a/1 **Implementation Note:** {*8652/0078*} {*AI95-00048-01*} The declaration of an RCI unit, with a calling-stub body, is automatically included in all active partitions with compilation units that depend on it. However the whole RCI library unit, including its (non-stub) body, will only be in one of the active partitions.

18 {*compilation units needed (remote call interface)* [partial]} {*needed (remote call interface)* [partial]} {*notwithstanding*} Notwithstanding the rule given in 10.2, a compilation unit in a given partition that semantically depends on the declaration of an RCI library unit, *needs* (in the sense of 10.2) only the declaration of the RCI library unit, not the body, to be included in that same partition. [Therefore, the

body of an RCI library unit is included only in the partition to which the RCI library unit is explicitly assigned.]

Implementation Requirements

19/1 {*8652/0078*} {*AI95-00048-01*} If a pragma All_Calls_Remote applies to a given RCI library unit, then the implementation shall route any call to a subprogram of the RCI unit from outside the declarative region of the unit through the Partition Communication Subsystem (PCS); see E.5. Calls to such subprograms from within the declarative region of the unit are defined to be local and shall not go through the PCS.

19.a/1 **Discussion:** {*8652/0078*} {*AI95-00048-01*} Without this pragma, it is presumed that most implementations will make direct calls if the call originates in the same partition as that of the RCI unit. With this pragma, all calls from outside the subsystem rooted at the RCI unit are treated like calls from outside the partition, ensuring that the PCS is involved in all such calls (for debugging, redundancy, etc.).

19.b **Reason:** There is no point to force local calls (or calls from children) to go through the PCS, since on the target system, these calls are always local, and all the units are in the same active partition.

Implementation Permissions

20 An implementation need not support the Remote_Call_Interface pragma nor the All_Calls_Remote pragma. [Explicit message-based communication between active partitions can be supported as an alternative to RPC.]

20.a **Ramification:** Of course, it is pointless to support the All_Calls_Remote pragma if the Remote_Call_Interface pragma (or some approximate equivalent) is not supported.

Incompatibilities With Ada 95

20.b/2 {*AI95-00240-01*} {*incompatibilities with Ada 95*} **Amendment Correction:** The wording was changed from “user-specified” to “available” attributes. (This was then further changed, see below.) This means that a type with the attributes specified in the private part would originally have been allowed as a formal parameter of an RCI subprogram, but that is no longer allowed. This change was made so that the rules were consistent with the rules introduced for the Corrigendum for stream attributes; moreover, legality should not depend on the contents of the private part.

Wording Changes from Ada 95

20.c/2 {*8652/0078*} {*AI95-00048-01*} **Corrigendum:** Changed the wording to allow a library subprogram to be a remote call interface unit.

20.d/2 {*AI95-00366-01*} Changed the wording to use the newly defined term *type that supports external streaming*, so that various issues with access types in pure units and implicitly declared attributes for type extensions are properly handled.

E.3 Consistency of a Distributed System

1 [This clause defines attributes and rules associated with verifying the consistency of a distributed program.]

Language Design Principles

1.a The rules guarantee that remote call interface and shared passive packages are consistent among all partitions prior to the execution of a distributed program, so that the semantics of the distributed program are well defined.

Static Semantics

2/1 For a prefix P that statically denotes a program unit, the following attributes are defined:

3 P'Version Yields a value of the predefined type String that identifies the version of the compilation unit that contains the declaration of the program unit.

4 P'Body_Version

Yields a value of the predefined type String that identifies the version of the compilation unit that contains the body (but not any subunits) of the program unit.

5/1 {*8652/0084*} {*AI95-00104-01*} {*version (of a compilation unit)*} The *version* of a compilation unit changes whenever the compilation unit changes in a semantically significant way. This International Standard does not define the exact meaning of "semantically significant". It is unspecified whether there are other events (such as recompilation) that result in the version of a compilation unit changing. {*unspecified* [partial]}

5.a/1 *This paragraph was deleted.*

5.1/1 {*8652/0084*} {*AI95-00104-01*} If P is not a library unit, and P has no completion, then P'Body_Version returns the Body_Version of the innermost program unit enclosing the declaration of P. If P is a library unit, and P has no completion, then P'Body_Version returns a value that is different from Body_Version of any version of P that has a completion.

Bounded (Run-Time) Errors

6 {*bounded error (cause)* [partial]} {*unit consistency*} In a distributed program, a library unit is *consistent* if the same version of its declaration is used throughout. It is a bounded error to elaborate a partition of a distributed program that contains a compilation unit that depends on a different version of the declaration of a shared passive or RCI library unit than that included in the partition to which the shared passive or RCI library unit was assigned. {*Program_Error (raised by failure of run-time check)*} As a result of this error, Program_Error can be raised in one or both partitions during elaboration; in any case, the partitions become inaccessible to one another.

6.a **Ramification:** Because a version changes if anything on which it depends undergoes a version change, requiring consistency for shared passive and remote call interface library units is sufficient to ensure consistency for the declared pure and remote types library units that define the types used for the objects and parameters through which interpartition communication takes place.

6.b Note that we do not require matching Body_Versions; it is irrelevant for shared passive and remote call interface packages, since only one copy of their body exists in a distributed program (in the absence of implicit replication), and we allow the bodies to differ for declared pure and remote types packages from partition to partition, presuming that the differences are due to required error corrections that took place during the execution of a long-running distributed program. The Body_Version attribute provides a means for performing stricter consistency checks.

Wording Changes from Ada 95

6.c/2 {*8652/0084*} {*AI95-00104-01*} **Corrigendum:** Clarified the meaning of 'Version and 'Body_Version.

E.4 Remote Subprogram Calls

1 {*remote subprogram call*} {*asynchronous remote procedure call* [distributed]} {*calling partition*} {*called partition*} {*remote subprogram binding*} A *remote subprogram call* is a subprogram call that invokes the execution of a subprogram in another partition. The partition that originates the remote subprogram call is the *calling partition*, and the partition that executes the corresponding subprogram body is the *called partition*. Some remote procedure calls are allowed to return prior to the completion of subprogram execution. These are called *asynchronous remote procedure calls*.

2 There are three different ways of performing a remote subprogram call:

3 - As a direct call on a (remote) subprogram explicitly declared in a remote call interface;

4 - As an indirect call through a value of a remote access-to-subprogram type;

5 - As a dispatching call with a controlling operand designated by a value of a remote access-to-class-wide type.

6 The first way of calling corresponds to a *static* binding between the calling and the called partition. The latter two ways correspond to a *dynamic* binding between the calling and the called partition.

7 A remote call interface library unit (see E.2.3) defines the remote subprograms or remote access types used for remote subprogram calls.

Language Design Principles

7.a Remote subprogram calls are standardized since the RPC paradigm is widely-used, and establishing an interface to it in the annex will increase the portability and reusability of distributed programs.

Legality Rules

8 In a dispatching call with two or more controlling operands, if one controlling operand is designated by a value of a remote access-to-class-wide type, then all shall be.

Dynamic Semantics

9 {*marshalling*} {*unmarshalling*} {*execution (remote subprogram call)* [partial]} For the execution of a remote subprogram call, subprogram parameters (and later the results, if any) are passed using a stream-oriented representation (see 13.13.1) [which is suitable for transmission between partitions]. This action is called *marshalling*. *Unmarshalling* is the reverse action of reconstructing the parameters or results from the stream-oriented representation. [Marshalling is performed initially as part of the remote subprogram call in the calling partition; unmarshalling is done in the called partition. After the remote subprogram completes, marshalling is performed in the called partition, and finally unmarshalling is done in the calling partition.]

10 {*calling stub*} {*receiving stub*} A *calling stub* is the sequence of code that replaces the subprogram body of a remotely called subprogram in the calling partition. A *receiving stub* is the sequence of code (the “wrapper”) that receives a remote subprogram call on the called partition and invokes the appropriate subprogram body.

10.a **Discussion:** The use of the term *stub* in this annex should not be confused with body_stub as defined in 10.1.3. The term *stub* is used here because it is a commonly understood term when talking about the RPC paradigm.

11 {*at-most-once execution*} Remote subprogram calls are executed at most once, that is, if the subprogram call returns normally, then the called subprogram's body was executed exactly once.

12 The task executing a remote subprogram call blocks until the subprogram in the called partition returns, unless the call is asynchronous. For an asynchronous remote procedure call, the calling task can become ready before the procedure in the called partition returns.

13 {*cancellation of a remote subprogram call*} If a construct containing a remote call is aborted, the remote subprogram call is *cancelled*. Whether the execution of the remote subprogram is immediately aborted as a result of the cancellation is implementation defined.

13.a **Implementation defined:** Whether the execution of the remote subprogram is immediately aborted as a result of cancellation.

14 If a remote subprogram call is received by a called partition before the partition has completed its elaboration, the call is kept pending until the called partition completes its elaboration (unless the call is cancelled by the calling partition prior to that).

15 If an exception is propagated by a remotely called subprogram, and the call is not an asynchronous call, the corresponding exception is reraised at the point of the remote subprogram call. For an asynchronous call, if the remote procedure call returns prior to the completion of the remotely called subprogram, any exception is lost.

16 The exception Communication_Error (see E.5) is raised if a remote call cannot be completed due to difficulties in communicating with the called partition.

17 {*potentially blocking operation (remote subprogram call)* [partial]} {*blocking, potentially (remote subprogram call)* [partial]} All forms of remote subprogram calls are potentially blocking operations (see 9.5.1).

17.a **Reason:** Asynchronous remote procedure calls are potentially blocking since the implementation may require waiting for the availability of shared resources to initiate the remote call.

18/1 {*8652/0085*} {*AI95-00215-01*} {*Accessibility_Check* [partial]} {*check, language-defined (Accessibility_Check)*} In a remote subprogram call with a formal parameter of a class-wide type, a check is made that the tag of the actual parameter identifies a tagged type declared in a declared-pure or shared passive library unit, or in the visible part of a remote types or remote call interface library unit. {*Program_Error (raised by failure of run-time check)*} Program_Error is raised if this check fails. In a remote function call which returns a class-wide type, the same check is made on the function result.

18.a/1 **Discussion:** {*8652/0085*} {*AI95-00215-01*} This check makes certain that the specific type passed or returned in an RPC satisfies the rules for a "communicable" type. Normally this is guaranteed by the compile-time restrictions on remote call interfaces. However, with class-wide types, it is possible to pass an object whose tag identifies a type declared outside the "safe" packages.

18.b This is considered an accessibility_check since only the types declared in "safe" packages are considered truly "global" (cross-partition). Other types are local to a single partition. This is analogous to the "accessibility" of global vs. local declarations in a single-partition program.

18.c This rule replaces a rule from an early version of Ada 9X which was given in the subclause on Remote Types Library Units (now E.2.2, “Remote Types Library Units”). That rule tried to prevent "bad" types from being sent by arranging for their tags to mismatch between partitions. However, that interfered with other uses of tags. The new rule allows tags to agree in all partitions, even for those types which are not "safe" to pass in an RPC.

19 {*Partition_Check* [partial]} {*check, language-defined (Partition_Check)*} In a dispatching call with two or more controlling operands that are designated by values of a remote access-to-class-wide type, a check is made [(in addition to the normal Tag_Check — see 11.5)] that all the remote access-to-class-wide values originated from Access attribute_references that were evaluated by tasks of the same active partition. {*Constraint_Error (raised by failure of run-time check)*} Constraint_Error is raised if this check fails.

19.a **Implementation Note:** When a remote access-to-class-wide value is created by an Access attribute_reference, the identity of the active partition that evaluated the attribute_reference should be recorded in the representation of the remote access value.

Implementation Requirements

20 The implementation of remote subprogram calls shall conform to the PCS interface as defined by the specification of the language-defined package System.RPC (see E.5). The calling stub shall use the Do_RPC procedure unless the remote procedure call is asynchronous in which case Do_APC shall be used. On the receiving side, the corresponding receiving stub shall be invoked by the RPC-receiver.

20.a **Implementation Note:** One possible implementation model is as follows:

20.b The code for calls to subprograms declared in an RCI package is generated normally, that is, the call-site is the same as for a local subprogram call. The code for the remotely callable subprogram bodies is also generated normally. Subprogram's prologue and epilogue are the same as for a local call.

20.c When compiling the specification of an RCI package, the compiler generates calling stubs for each visible subprogram. Similarly, when compiling the body of an RCI package, the compiler generates receiving stubs for each visible subprogram together with the appropriate tables to allow the RPC-receiver to locate the correct receiving stub.

20.d For the statically bound remote calls, the identity of the remote partition is statically determined (it is resolved at configuration/link time).

20.e The calling stub operates as follows:

20.f
- It allocates (or reuses) a stream of Params_Stream_Type of Initial_Size, and initializes it by repeatedly calling Write operations, first to identify which remote subprogram in the receiving partition is being called, and then to pass the incoming value of each of the **in** and **in out** parameters of the call.

20.g
- It allocates (or reuses) a stream for the Result, unless a pragma Asynchronous is applied to the procedure.

- It calls Do_RPC unless a pragma Asynchronous is applied to the procedure in which case it calls Do_APC. An access value designating the message stream allocated and initialized above is passed as the Params parameter. An access value designating the Result stream is passed as the Result parameter. 20.h
- If the pragma Asynchronous is not specified for the procedure, Do_RPC blocks until a reply message arrives, and then returns to the calling stub. The stub returns after extracting from the Result stream, using Read operations, the **in out** and **out** parameters or the function result. If the reply message indicates that the execution of the remote subprogram propagated an exception, the exception is propagated from Do_RPC to the calling stub, and thence to the point of the original remote subprogram call. If Do_RPC detects that communication with the remote partition has failed, it propagates Communication_Error. 20.i

On the receiving side, the RPC-receiver procedure operates as follows: 20.j

- It is called from the PCS when a remote-subprogram-call message is received. The call originates in some remote call receiver task executed and managed in the context of the PCS. 20.k
- It extracts information from the stream to identify the appropriate receiving stub. 20.l
- The receiving stub extracts the **in** and **in out** parameters using Read from the stream designated by the Params parameter. 20.m
- The receiving stub calls the actual subprogram body and, upon completion of the subprogram, uses Write to insert the results into the stream pointed to by the Result parameter. The receiving stub returns to the RPC-receiver procedure which in turn returns to the PCS. If the actual subprogram body propagates an exception, it is propagated by the RPC-receiver to the PCS, which handles the exception, and indicates in the reply message that the execution of the subprogram body propagated an exception. The exception occurrence can be represented in the reply message using the Write attribute of Ada.Exceptions.Exception_Occurrence. 20.n

For remote access-to-subprogram types: 20.o

A value of a remote access-to-subprogram type can be represented by the following components: a reference to the remote partition, an index to the package containing the remote subprogram, and an index to the subprogram within the package. The values of these components are determined at run time when the remote access value is created. These three components serve the same purpose when calling Do_APC/RPC, as in the statically bound remote calls; the only difference is that they are evaluated dynamically. 20.p

For remote access-to-class-wide types: 20.q

For each remote access-to-class-wide type, a calling stub is generated for each dispatching operation of the designated type. In addition, receiving stubs are generated to perform the remote dispatching operations in the called partition. The appropriate subprogram_body is determined as for a local dispatching call once the receiving stub has been reached. 20.r

A value of a remote access-to-class-wide type can be represented with the following components: a reference to the remote partition, an index to a table (created one per each such access type) containing addresses of all the dispatching operations of the designated type, and an access value designating the actual remote object. 20.s

Alternatively, a remote access-to-class-wide value can be represented as a normal access value, pointing to a "stub" object which in turn contains the information mentioned above. A call on any dispatching operation of such a stub object does the remote call, if necessary, using the information in the stub object to locate the target partition, etc. This approach has the advantage that less special-casing is required in the compiler. All access values can remain just a simple address. 20.t

{*Constraint_Error (raised by failure of run-time check)*} For a call to Do_RPC or Do_APC: The partition ID of all controlling operands are checked for equality (a Constraint_Error is raised if this check fails). The partition ID value is used for the Partition parameter. An index into the *tagged-type-descriptor* is created. This index points to the receiving stub of the class-wide operation. This index and the index to the table (described above) are written to the stream. Then, the actual parameters are marshalled into the message stream. For a controlling operand, only the access value designating the remote object is required (the other two components are already present in the other parameters). 20.u

On the called partition (after the RPC-receiver has transferred control to the appropriate receiving stub) the parameters are first unmarshalled. Then, the tags of the controlling operands (obtained by dereferencing the pointer to the object) are checked for equality. {*Constraint_Error (raised by failure of run-time check)*} If the check fails Constraint_Error is raised and propagated back to the calling partition, unless it is a result of an asynchronous call. Finally, a dispatching call to the specific subprogram (based on the controlling object's tag) is made. Note that since this subprogram is not in an RCI package, no specific stub is generated for it, it is called normally from the *dispatching stub*. 20.v

{*8652/0086*} {*AI95-00159-01*} With respect to shared variables in shared passive library units, the execution of the corresponding subprogram body of a synchronous remote procedure call is considered to be part of the execution of the calling task. The execution of the corresponding subprogram body of an asynchronous remote procedure call proceeds in parallel with the calling task and does not signal the next action of the calling task (see 9.10). 20.1/1

NOTES

21 6 A given active partition can both make and receive remote subprogram calls. Thus, an active partition can act as both a client and a server.

22 7 If a given exception is propagated by a remote subprogram call, but the exception does not exist in the calling partition, the exception can be handled by an **others** choice or be propagated to and handled by a third partition.

22.a **Discussion:** This situation can happen in a case of dynamically nested remote subprogram calls, where an intermediate call executes in a partition that does not include the library unit that defines the exception.

Wording Changes from Ada 95

22.b/2 {*8652/0086*} {*AI95-00159-01*} **Corrigendum:** Added rules so that tasks can safely access shared passive objects.

22.c/2 {*8652/0085*} {*AI95-00215-01*} **Corrigendum:** Clarified that the check on class-wide types also applies to values returned from remote subprogram call functions.

E.4.1 Pragma Asynchronous

1 [This subclause introduces the pragma Asynchronous which allows a remote subprogram call to return prior to completion of the execution of the corresponding remote subprogram body.]

Syntax

2 The form of a pragma Asynchronous is as follows:

3 **pragma** Asynchronous(local_name);

Legality Rules

4 The local_name of a pragma Asynchronous shall denote either:

5 • One or more remote procedures; the formal parameters of the procedure(s) shall all be of mode **in**;

6 • The first subtype of a remote access-to-procedure type; the formal parameters of the designated profile of the type shall all be of mode **in**;

7 • The first subtype of a remote access-to-class-wide type.

Static Semantics

8 {*representation pragma (Asynchronous)* [partial]} {*pragma, representation (Asynchronous)* [partial]} A pragma Asynchronous is a representation pragma. When applied to a type, it specifies the type-related *asynchronous* aspect of the type.

Dynamic Semantics

9 {*remote procedure call (asynchronous)*} {*asynchronous (remote procedure call)*} A remote call is *asynchronous* if it is a call to a procedure, or a call through a value of an access-to-procedure type, to which a pragma Asynchronous applies. In addition, if a pragma Asynchronous applies to a remote access-to-class-wide type, then a dispatching call on a procedure with a controlling operand designated by a value of the type is asynchronous if the formal parameters of the procedure are all of mode **in**.

Implementation Requirements

10 Asynchronous remote procedure calls shall be implemented such that the corresponding body executes at most once as a result of the call.

10.a **To be honest:** It is not clear that this rule can be tested or even defined formally.

E.4.2 Example of Use of a Remote Access-to-Class-Wide Type

Examples

Example of using a remote access-to-class-wide type to achieve dynamic binding across active partitions: 1

2
```
package Tapes is
   pragma Pure(Tapes);
   type Tape is abstract tagged limited private;
   -- Primitive dispatching operations where
   -- Tape is controlling operand
   procedure Copy (From, To : access Tape; Num_Recs : in Natural) is abstract;
   procedure Rewind (T : access Tape) is abstract;
   -- More operations
private
   type Tape is ...
end Tapes;
```

3
```
with Tapes;
package Name_Server is
   pragma Remote_Call_Interface;
   -- Dynamic binding to remote operations is achieved
   -- using the access-to-limited-class-wide type Tape_Ptr
   type Tape_Ptr is access all Tapes.Tape'Class;
   -- The following statically bound remote operations
   -- allow for a name-server capability in this example
   function  Find     (Name : String) return Tape_Ptr;
   procedure Register (Name : in String; T : in Tape_Ptr);
   procedure Remove   (T : in Tape_Ptr);
   -- More operations
end Name_Server;
```

4
```
package Tape_Driver is
  -- Declarations are not shown, they are irrelevant here
end Tape_Driver;
```

5
```
with Tapes, Name_Server;
package body Tape_Driver is
   type New_Tape is new Tapes.Tape with ...
   procedure Copy
    (From, To : access New_Tape; Num_Recs: in Natural) is
   begin
     . . .
   end Copy;
   procedure Rewind (T : access New_Tape) is
   begin
      . . .
   end Rewind;
   -- Objects remotely accessible through use
   -- of Name_Server operations
   Tape1, Tape2 : aliased New_Tape;
begin
   Name_Server.Register ("NINE-TRACK",  Tape1'Access);
   Name_Server.Register ("SEVEN-TRACK", Tape2'Access);
end Tape_Driver;
```

6
```
with Tapes, Name_Server;
-- Tape_Driver is not needed and thus not mentioned in the with_clause
procedure Tape_Client is
   T1, T2 : Name_Server.Tape_Ptr;
begin
   T1 := Name_Server.Find ("NINE-TRACK");
   T2 := Name_Server.Find ("SEVEN-TRACK");
   Tapes.Rewind (T1);
   Tapes.Rewind (T2);
   Tapes.Copy (T1, T2, 3);
end Tape_Client;
```

7 *Notes on the example*:

7.a **Discussion:** The example does not show the case where tapes are removed from or added to the system. In the former case, an appropriate exception needs to be defined to instruct the client to use another tape. In the latter, the Name_Server should have a query function visible to the clients to inform them about the availability of the tapes in the system.

8/1 *This paragraph was deleted.*

9 • The package Tapes provides the necessary declarations of the type and its primitive operations.

10 • Name_Server is a remote call interface package and is elaborated in a separate active partition to provide the necessary naming services (such as Register and Find) to the entire distributed program through remote subprogram calls.

11 • Tape_Driver is a normal package that is elaborated in a partition configured on the processing node that is connected to the tape device(s). The abstract operations are overridden to support the locally declared tape devices (Tape1, Tape2). The package is not visible to its clients, but it exports the tape devices (as remote objects) through the services of the Name_Server. This allows for tape devices to be dynamically added, removed or replaced without requiring the modification of the clients' code.

12 • The Tape_Client procedure references only declarations in the Tapes and Name_Server packages. Before using a tape for the first time, it needs to query the Name_Server for a system-wide identity for that tape. From then on, it can use that identity to access the tape device.

13 • Values of remote access type Tape_Ptr include the necessary information to complete the remote dispatching operations that result from dereferencing the controlling operands T1 and T2.

E.5 Partition Communication Subsystem

1/2 {*AI95-00273-01*} {*partition communication subsystem (PCS)*} {*PCS (partition communication subsystem)*} [The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS.]

1.a **Reason:** The prefix RPC is used rather than RSC because the term remote procedure call and its acronym are more familiar.

Static Semantics

2 The following language-defined library package exists:

3
```
with Ada.Streams; -- see 13.13.1
package System.RPC is
```

4
```
   type Partition_Id is range 0 .. implementation-defined;
```

5
```
   Communication_Error : exception;
```

6
```
   type Params_Stream_Type (
      Initial_Size : Ada.Streams.Stream_Element_Count) is new
      Ada.Streams.Root_Stream_Type with private;
```

7
```
   procedure Read(
      Stream : in out Params_Stream_Type;
      Item : out Ada.Streams.Stream_Element_Array;
      Last : out Ada.Streams.Stream_Element_Offset);
```

8
```
   procedure Write(
      Stream : in out Params_Stream_Type;
      Item : in Ada.Streams.Stream_Element_Array);
      -- Synchronous call                                          9
      procedure Do_RPC(
         Partition  : in Partition_Id;
         Params     : access Params_Stream_Type;
         Result     : access Params_Stream_Type);

      -- Asynchronous call                                         10
      procedure Do_APC(
         Partition  : in Partition_Id;
         Params     : access Params_Stream_Type);

      -- The handler for incoming RPCs                             11
      type RPC_Receiver is access procedure(
         Params     : access Params_Stream_Type;
         Result     : access Params_Stream_Type);

      procedure Establish_RPC_Receiver(                            12
         Partition : in Partition_Id;
         Receiver  : in RPC_Receiver);

   private                                                         13
      ... -- not specified by the language
   end System.RPC;
```

A value of the type Partition_Id is used to identify a partition. 14

> **Implementation defined:** The range of type System.RPC.Partition_Id. 14.a/2

An object of the type Params_Stream_Type is used for identifying the particular remote subprogram that is being called, as well as marshalling and unmarshalling the parameters or result of a remote subprogram call, as part of sending them between partitions. 15

[The Read and Write procedures override the corresponding abstract operations for the type Params_Stream_Type.] 16

Dynamic Semantics

The Do_RPC and Do_APC procedures send a message to the active partition identified by the Partition parameter. 17

> **Implementation Note:** It is assumed that the RPC interface is above the message-passing layer of the network protocol stack and is implemented in terms of it. 17.a

After sending the message, Do_RPC blocks the calling task until a reply message comes back from the called partition or some error is detected by the underlying communication system in which case Communication_Error is raised at the point of the call to Do_RPC. 18

> **Reason:** Only one exception is defined in System.RPC, although many sources of errors might exist. This is so because it is not always possible to distinguish among these errors. In particular, it is often impossible to tell the difference between a failing communication link and a failing processing node. Additional information might be associated with a particular Exception_Occurrence for a Communication_Error. 18.a

Do_APC operates in the same way as Do_RPC except that it is allowed to return immediately after sending the message. 19

Upon normal return, the stream designated by the Result parameter of Do_RPC contains the reply message. 20

{*elaboration (partition)* [partial]} The procedure System.RPC.Establish_RPC_Receiver is called once, immediately after elaborating the library units of an active partition (that is, right after the *elaboration of the partition*) if the partition includes an RCI library unit, but prior to invoking the main subprogram, if any. The Partition parameter is the Partition_Id of the active partition being elaborated. {*RPC-receiver*} The Receiver parameter designates an implementation-provided procedure called the *RPC-receiver* which will handle all RPCs received by the partition from the PCS. Establish_RPC_Receiver saves a reference to the RPC-receiver; when a message is received at the called partition, the RPC-receiver is called with 21 the Params stream containing the message. When the RPC-receiver returns, the contents of the stream designated by Result is placed in a message and sent back to the calling partition.

21.a **Implementation Note:** It is defined by the PCS implementation whether one or more threads of control should be available to process incoming messages and to wait for their completion.

21.b **Implementation Note:** At link-time, the linker provides the RPC-receiver and the necessary tables to support it. A call on Establish_RPC_Receiver is inserted just before the call on the main subprogram.

21.c **Reason:** The interface between the PCS (the System.RPC package) and the RPC-receiver is defined to be dynamic in order to allow the elaboration sequence to notify the PCS that all packages have been elaborated and that it is safe to call the receiving stubs. It is not guaranteed that the PCS units will be the last to be elaborated, so some other indication that elaboration is complete is needed.

22 If a call on Do_RPC is aborted, a cancellation message is sent to the called partition, to request that the execution of the remotely called subprogram be aborted.

22.a **To be honest:** The full effects of this message are dependent on the implementation of the PCS.

23 {*potentially blocking operation (RPC operations)* [partial]} {*blocking, potentially (RPC operations)* [partial]} The subprograms declared in System.RPC are potentially blocking operations.

Implementation Requirements

24 The implementation of the RPC-receiver shall be reentrant[, thereby allowing concurrent calls on it from the PCS to service concurrent remote subprogram calls into the partition].

24.a **Reason:** There seems no reason to allow the implementation of RPC-receiver to be nonreentrant, even though we don't require that every implementation of the PCS actually perform concurrent calls on the RPC-receiver.

24.1/1 {*8652/0087*} {*AI95-00082-01*} An implementation shall not restrict the replacement of the body of System.RPC. An implementation shall not restrict children of System.RPC. [The related implementation permissions in the introduction to Annex A do not apply.]

24.a.1/1 **Reason:** The point of System.RPC is to let the user tailor the communications mechanism without requiring changes to or other cooperation from the compiler. However, implementations can restrict the replacement of language-defined units. This requirement overrides that permission for System.RPC.

24.2/1 {*8652/0087*} {*AI95-00082-01*} If the implementation of System.RPC is provided by the user, an implementation shall support remote subprogram calls as specified.

24.b/2 **Discussion:** {*AI95-00273-01*} If the implementation takes advantage of the implementation permission to use a different specification for System.RPC, it still needs to use it for remote subprogram calls, and allow the user to replace the body of System.RPC. It just isn't guaranteed to be portable to do so in Ada 2005 - an advantage which was more theoretical than real anyway.

Documentation Requirements

25 The implementation of the PCS shall document whether the RPC-receiver is invoked from concurrent tasks. If there is an upper limit on the number of such tasks, this limit shall be documented as well, together with the mechanisms to configure it (if this is supported).

25.a/2 *This paragraph was deleted.*

25.a.1/2 **Documentation Requirement:** Whether the RPC-receiver is invoked from concurrent tasks, and if so, the number of such tasks.

Implementation Permissions

26 The PCS is allowed to contain implementation-defined interfaces for explicit message passing, broadcasting, etc. Similarly, it is allowed to provide additional interfaces to query the state of some remote partition (given its partition ID) or of the PCS itself, to set timeouts and retry parameters, to get more detailed error status, etc. These additional interfaces should be provided in child packages of System.RPC.

26.a **Implementation defined:** Implementation-defined interfaces in the PCS.

A body for the package System.RPC need not be supplied by the implementation. 27

Reason: It is presumed that a body for the package System.RPC might be extremely environment specific. Therefore, we do not require that a body be provided by the (compiler) implementation. The user will have to write a body, or acquire one, appropriate for the target environment. 27.a

{*AI95-00273-01*} An alternative declaration is allowed for package System.RPC as long as it provides a set of operations that is substantially equivalent to the specification defined in this clause. 27.1/2

Reason: Experience has proved that the definition of System.RPC given here is inadequate for interfacing to existing distribution mechanisms (such as CORBA), especially on heterogeneous systems. Rather than mandate a change in the mechanism (which would break existing systems), require implementations to support multiple mechanisms (which is impractical), or prevent the use of Annex E facilities with existing systems (which would be silly), we simply make this facility optional. 27.b/2

One of the purposes behind System.RPC was that knowledgeable users, rather than compiler vendors, could create this package tailored to their networks. Experience has shown that users get their RPC from vendors anyway; users have not taken advantage of the flexibility provided by this defined interface. Moreover, one could compare this defined interface to requiring Ada compilers to use a defined interface to implement tasking. No one thinks that the latter is a good idea, why should anyone believe that the former is? 27.c/2

Therefore, this clause is made optional. We considered deleting the clause outright, but we still require that users may replace the package (whatever its interface). Also, it still provides a useful guide to the implementation of this feature. 27.d/2

Implementation Advice

Whenever possible, the PCS on the called partition should allow for multiple tasks to call the RPC-receiver with different messages and should allow them to block until the corresponding subprogram body returns. 28

Implementation Advice: The PCS should allow for multiple tasks to call the RPC-receiver. 28.a/2

The Write operation on a stream of type Params_Stream_Type should raise Storage_Error if it runs out of space trying to write the Item into the stream. 29

Implementation Advice: The System.RPC.Write operation should raise Storage_Error if it runs out of space when writing an item. 29.a.1/2

Implementation Note: An implementation could also dynamically allocate more space as needed, only propagating Storage_Error if the allocator it calls raises Storage_Error. This storage could be managed through a controlled component of the stream object, to ensure that it is reclaimed when the stream object is finalized. 29.a

NOTES

8 The package System.RPC is not designed for direct calls by user programs. It is instead designed for use in the implementation of remote subprograms calls, being called by the calling stubs generated for a remote call interface library unit to initiate a remote call, and in turn calling back to an RPC-receiver that dispatches to the receiving stubs generated for the body of a remote call interface, to handle a remote call received from elsewhere. 30

Incompatibilities With Ada 95

{*AI95-00273-01*} {*incompatibilities with Ada 95*} The specification of System.RPC can now be tailored for an implementation. If a program replaces the body of System.RPC with a user-defined body, it might not compile in a given implementation of Ada 2005 (if the specification of System.RPC has been changed). 30.a/2

Wording Changes from Ada 95

{*8652/0087*} {*AI95-00082-01*} **Corrigendum:** Clarified that the user can replace System.RPC. 30.b/2

Annex F
(normative)
Information Systems

{*information systems*} This Annex provides a set of facilities relevant to Information Systems programming. These fall into several categories: 1

- an attribute definition clause specifying Machine_Radix for a decimal subtype; 2
- the package Decimal, which declares a set of constants defining the implementation's capacity for decimal types, and a generic procedure for decimal division; and 3
- {*AI95-00285-01*} the child packages Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing, which support formatted and localized output of decimal data, based on “picture String” values. 4/2

{*AI95-00434-01*} See also: 3.5.9, “Fixed Point Types”; 3.5.10, “Operations of Fixed Point Types”; 4.6, “Type Conversions”; 13.3, “Operational and Representation Attributes”; A.10.9, “Input-Output for Real Types”; B.3, “Interfacing with C and C++”; B.4, “Interfacing with COBOL”; Annex G, “Numerics”. 5/2

The character and string handling packages in Annex A, “Predefined Language Environment” are also relevant for Information Systems. 6

Implementation Advice

If COBOL (respectively, C) is widely supported in the target environment, implementations supporting the Information Systems Annex should provide the child package Interfaces.COBOL (respectively, Interfaces.C) specified in Annex B and should support a *convention_*identifier of COBOL (respectively, C) in the interfacing pragmas (see Annex B), thus allowing Ada programs to interface with programs written in that language. 7

Implementation Advice: If COBOL (respectively, C) is supported in the target environment, then interfacing to COBOL (respectively, C) should be supported as specified in Annex B. 7.a/2

Extensions to Ada 83

{*extensions to Ada 83*} This Annex is new to Ada 95. 7.b

Wording Changes from Ada 95

{*AI95-00285-01*} Added a mention of Wide_Wide_Text_IO.Editing, part of the support for 32-bit characters. 7.c/2

F.1 Machine_Radix Attribute Definition Clause

Static Semantics

{*specifiable (of Machine_Radix for decimal first subtypes)* [partial]} {*Machine_Radix clause*} Machine_Radix may be specified for a decimal first subtype (see 3.5.9) via an attribute_definition_clause; the expression of such a clause shall be static, and its value shall be 2 or 10. A value of 2 implies a binary base range; a value of 10 implies a decimal base range. 1

Ramification: In the absence of a Machine_Radix clause, the choice of 2 versus 10 for S'Machine_Radix is not specified. 1.a

Implementation Advice

Packed decimal should be used as the internal representation for objects of subtype S when S'Machine_Radix = 10. 2

2.a/2 **Implementation Advice:** Packed decimal should be used as the internal representation for objects of subtype *S* when *S*'Machine_Radix = 10.

2.b **Discussion:** The intent of a decimal Machine_Radix attribute definition clause is to allow the programmer to declare an Ada decimal data object whose representation matches a particular COBOL implementation's representation of packed decimal items. The Ada object may then be passed to an interfaced COBOL program that takes a packed decimal data item as a parameter, assuming that convention COBOL has been specified for the Ada object's type in a pragma Convention.

2.c Additionally, the Ada compiler may choose to generate arithmetic instructions that exploit the packed decimal representation.

Examples

3 *Example of Machine_Radix attribute definition clause:*

4
```
type Money is delta 0.01 digits 15;
for Money'Machine_Radix use 10;
```

F.2 The Package Decimal

Static Semantics

1 The library package Decimal has the following declaration:

2
```
package Ada.Decimal is
   pragma Pure(Decimal);
```
3
```
   Max_Scale : constant := implementation-defined;
   Min_Scale : constant := implementation-defined;
```
4
```
   Min_Delta : constant := 10.0**(-Max_Scale);
   Max_Delta : constant := 10.0**(-Min_Scale);
```
5
```
   Max_Decimal_Digits : constant := implementation-defined;
```
6
```
   generic
      type Dividend_Type  is delta <> digits <>;
      type Divisor_Type   is delta <> digits <>;
      type Quotient_Type  is delta <> digits <>;
      type Remainder_Type is delta <> digits <>;
   procedure Divide (Dividend  : in Dividend_Type;
                     Divisor   : in Divisor_Type;
                     Quotient  : out Quotient_Type;
                     Remainder : out Remainder_Type);
   pragma Convention(Intrinsic, Divide);
```
7
```
end Ada.Decimal;
```

7.a **Implementation defined:** The values of named numbers in the package Decimal.

8 Max_Scale is the largest N such that 10.0**(–N) is allowed as a decimal type's delta. Its type is *universal_integer*.

9 Min_Scale is the smallest N such that 10.0**(–N) is allowed as a decimal type's delta. Its type is *universal_integer*.

10 Min_Delta is the smallest value allowed for *delta* in a decimal_fixed_point_definition. Its type is *universal_real*.

11 Max_Delta is the largest value allowed for *delta* in a decimal_fixed_point_definition. Its type is *universal_real*.

12 Max_Decimal_Digits is the largest value allowed for *digits* in a decimal_fixed_point_definition. Its type is *universal_integer*.

12.a **Reason:** The name is Max_Decimal_Digits versus Max_Digits, in order to avoid confusion with the named number System.Max_Digits relevant to floating point.

Static Semantics

The effect of Divide is as follows. The value of Quotient is Quotient_Type(Dividend/Divisor). The value of Remainder is Remainder_Type(Intermediate), where Intermediate is the difference between Dividend and the product of Divisor and Quotient; this result is computed exactly. 13

Implementation Requirements

Decimal.Max_Decimal_Digits shall be at least 18. 14

Decimal.Max_Scale shall be at least 18. 15

Decimal.Min_Scale shall be at most 0. 16

NOTES

1 The effect of division yielding a quotient with control over rounding versus truncation is obtained by applying either the function attribute Quotient_Type'Round or the conversion Quotient_Type to the expression Dividend/Divisor. 17

F.3 Edited Output for Decimal Types

{*AI95-00285-01*} The child packages Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing provide localizable formatted text output, known as *edited output*{*edited output*} , for decimal types. An edited output string is a function of a numeric value, program-specifiable locale elements, and a format control value. The numeric value is of some decimal type. The locale elements are: 1/2

- the currency string; 2
- the digits group separator character; 3
- the radix mark character; and 4
- the fill character that replaces leading zeros of the numeric value. 5

{*AI95-00285-01*} For Text_IO.Editing the edited output and currency strings are of type String, and the locale characters are of type Character. For Wide_Text_IO.Editing their types are Wide_String and Wide_Character, respectively. For Wide_Wide_Text_IO.Editing their types are Wide_Wide_String and Wide_Wide_Character, respectively. 6/2

Each of the locale elements has a default value that can be replaced or explicitly overridden. 7

A format-control value is of the private type Picture; it determines the composition of the edited output string and controls the form and placement of the sign, the position of the locale elements and the decimal digits, the presence or absence of a radix mark, suppression of leading zeros, and insertion of particular character values. 8

A Picture object is composed from a String value, known as a *picture String*, that serves as a template for the edited output string, and a Boolean value that controls whether a string of all space characters is produced when the number's value is zero. A picture String comprises a sequence of one- or two-Character symbols, each serving as a placeholder for a character or string at a corresponding position in the edited output string. The picture String symbols fall into several categories based on their effect on the edited output string: 9

10

Decimal Digit:	'9'					
Radix Control:	'.'	'V'				
Sign Control:	'+'	'–'	'<'	'>'	"CR"	"DB"
Currency Control:	'$'	'#'				
Zero Suppression:	'Z'	'*'				
Simple Insertion:	'_'	'B'	'0'	'/'		

11 The entries are not case-sensitive. Mixed- or lower-case forms for "CR" and "DB", and lower-case forms for 'V', 'Z', and 'B', have the same effect as the upper-case symbols shown.

12 An occurrence of a '9' Character in the picture String represents a decimal digit position in the edited output string.

13 A radix control Character in the picture String indicates the position of the radix mark in the edited output string: an actual character position for '.', or an assumed position for 'V'.

14 A sign control Character in the picture String affects the form of the sign in the edited output string. The '<' and '>' Character values indicate parentheses for negative values. A Character '+', '–', or '<' appears either singly, signifying a fixed-position sign in the edited output, or repeated, signifying a floating-position sign that is preceded by zero or more space characters and that replaces a leading 0.

15 A currency control Character in the picture String indicates an occurrence of the currency string in the edited output string. The '$' Character represents the complete currency string; the '#' Character represents one character of the currency string. A '$' Character appears either singly, indicating a fixed-position currency string in the edited output, or repeated, indicating a floating-position currency string that occurs in place of a leading 0. A sequence of '#' Character values indicates either a fixed- or floating-position currency string, depending on context.

16 A zero suppression Character in the picture String allows a leading zero to be replaced by either the space character (for 'Z') or the fill character (for '*').

17 A simple insertion Character in the picture String represents, in general, either itself (if '/' or '0'), the space character (if 'B'), or the digits group separator character (if '_'). In some contexts it is treated as part of a floating sign, floating currency, or zero suppression string.

18/2 {*AI95-00434-01*} An example of a picture String is "<###Z_ZZ9.99>". If the currency string is "kr", the separator character is ',', and the radix mark is '.' then the edited output string values for the decimal values 32.10 and –5432.10 are "bbkrbbb32.10b" and "(bkr5,432.10)", respectively, where 'b' indicates the space character.

19/2 {*AI95-00285-01*} The generic packages Text_IO.Decimal_IO, Wide_Text_IO.Decimal_IO, and Wide_Wide_Text_IO.Decimal_IO (see A.10.9, "Input-Output for Real Types") provide text input and non-edited text output for decimal types.

NOTES

20/2 2 {*AI95-00285-01*} A picture String is of type Standard.String, for all of Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing.

Wording Changes from Ada 95

20.a/2 {*AI95-00285-01*} Added descriptions of Wide_Wide_Text_IO.Editing; see F.3.5.

F.3.1 Picture String Formation

1 {*picture String (for edited output)*} {*well-formed picture String (for edited output)*} A *well-formed picture String*, or simply *picture String*, is a String value that conforms to the syntactic rules, composition constraints, and character replication conventions specified in this clause.

Dynamic Semantics

2/1 *This paragraph was deleted.*

```
picture_string ::=                                                              3
  fixed_$_picture_string
 | fixed_#_picture_string
 | floating_currency_picture_string
 | non_currency_picture_string

fixed_$_picture_string ::=                                                      4
  [fixed_LHS_sign] fixed_$_char {direct_insertion} [zero_suppression]
    number [RHS_sign]

 | [fixed_LHS_sign {direct_insertion}] [zero_suppression]
    number fixed_$_char {direct_insertion} [RHS_sign]

 | floating_LHS_sign number fixed_$_char {direct_insertion} [RHS_sign]

 | [fixed_LHS_sign] fixed_$_char {direct_insertion}
    all_zero_suppression_number {direct_insertion}  [RHS_sign]

 | [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
    fixed_$_char {direct_insertion} [RHS_sign]

 | all_sign_number {direct_insertion} fixed_$_char {direct_insertion} [RHS_sign]

fixed_#_picture_string ::=                                                      5
  [fixed_LHS_sign] single_#_currency {direct_insertion}
    [zero_suppression] number [RHS_sign]

 | [fixed_LHS_sign] multiple_#_currency {direct_insertion}
    zero_suppression number [RHS_sign]

 | [fixed_LHS_sign {direct_insertion}] [zero_suppression]
    number fixed_#_currency {direct_insertion} [RHS_sign]

 | floating_LHS_sign number fixed_#_currency {direct_insertion} [RHS_sign]

 | [fixed_LHS_sign] single_#_currency {direct_insertion}
    all_zero_suppression_number {direct_insertion} [RHS_sign]

 | [fixed_LHS_sign] multiple_#_currency {direct_insertion}
    all_zero_suppression_number {direct_insertion} [RHS_sign]

 | [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
    fixed_#_currency {direct_insertion} [RHS_sign]

 | all_sign_number {direct_insertion} fixed_#_currency {direct_insertion} [RHS_sign]

floating_currency_picture_string ::=                                            6
  [fixed_LHS_sign] {direct_insertion} floating_$_currency number [RHS_sign]
 | [fixed_LHS_sign] {direct_insertion} floating_#_currency number [RHS_sign]
 | [fixed_LHS_sign] {direct_insertion} all_currency_number {direct_insertion} [RHS_sign]
```

7 non_currency_picture_string ::=
  [fixed_LHS_sign {direct_insertion}] zero_suppression number [RHS_sign]
 | [floating_LHS_sign] number [RHS_sign]
 | [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
   [RHS_sign]
 | all_sign_number {direct_insertion}
 | fixed_LHS_sign direct_insertion {direct_insertion} number [RHS_sign]

8 fixed_LHS_sign ::= LHS_Sign

9 LHS_Sign ::= + | – | <

10 fixed_$_char ::= $

11 direct_insertion ::= simple_insertion

12 simple_insertion ::= _ | B | 0 | /

13 zero_suppression ::= Z {Z | context_sensitive_insertion} | fill_string

14 context_sensitive_insertion ::= simple_insertion

15 fill_string ::= * {* | context_sensitive_insertion}

16 number ::=
  fore_digits [radix [aft_digits] {direct_insertion}]
 | radix aft_digits {direct_insertion}

17 fore_digits ::= 9 {9 | direct_insertion}

18 aft_digits ::= {9 | direct_insertion} 9

19 radix ::= . | V

20 RHS_sign ::= + | – | > | CR | DB

21 floating_LHS_sign ::=
  LHS_Sign {context_sensitive_insertion} LHS_Sign {LHS_Sign | context_sensitive_insertion}

22 single_#_currency ::= #

23 multiple_#_currency ::= ## {#}

24 fixed_#_currency ::= single_#_currency | multiple_#_currency

25 floating_$_currency ::=
  $ {context_sensitive_insertion} $ {$ | context_sensitive_insertion}

26 floating_#_currency ::=
  # {context_sensitive_insertion} # {# | context_sensitive_insertion}

27 all_sign_number ::= all_sign_fore [radix [all_sign_aft]] [>]

28
```
all_sign_fore ::=
  sign_char {context_sensitive_insertion} sign_char {sign_char | context_sensitive_insertion}
```

29
```
all_sign_aft ::= {all_sign_aft_char} sign_char

all_sign_aft_char ::=  sign_char | context_sensitive_insertion
```

30
```
sign_char ::= + | – | <
```

31
```
all_currency_number ::=  all_currency_fore [radix [all_currency_aft]]
```

32
```
all_currency_fore ::=
  currency_char {context_sensitive_insertion}
    currency_char {currency_char | context_sensitive_insertion}
```

33
```
all_currency_aft ::= {all_currency_aft_char} currency_char

all_currency_aft_char ::= currency_char | context_sensitive_insertion
```

34
```
currency_char ::= $ | #
```

35
```
all_zero_suppression_number ::=  all_zero_suppression_fore [ radix [all_zero_suppression_aft]]
```

36
```
all_zero_suppression_fore ::=
  zero_suppression_char {zero_suppression_char | context_sensitive_insertion}
```

37
```
all_zero_suppression_aft ::= {all_zero_suppression_aft_char} zero_suppression_char

all_zero_suppression_aft_char ::=  zero_suppression_char | context_sensitive_insertion
```

38
```
zero_suppression_char ::= Z | *
```

39 The following composition constraints apply to a picture String:

- 40 A floating_LHS_sign does not have occurrences of different LHS_Sign Character values.
- 41 If a picture String has '<' as fixed_LHS_sign, then it has '>' as RHS_sign.
- 42 If a picture String has '<' in a floating_LHS_sign or in an all_sign_number, then it has an occurrence of '>'.
- 43/1 {*8652/0088*} {*AI95-00153*} If a picture String has '+' or '–' as fixed_LHS_sign, in a floating_LHS_sign, or in an all_sign_number, then it has no RHS_sign or '>' character.
- 44 An instance of all_sign_number does not have occurrences of different sign_char Character values.
- 45 An instance of all_currency_number does not have occurrences of different currency_char Character values.
- 46 An instance of all_zero_suppression_number does not have occurrences of different zero_-suppression_char Character values, except for possible case differences between 'Z' and 'z'.

47 A *replicable Character* is a Character that, by the above rules, can occur in two consecutive positions in a picture String.

48 A *Character replication* is a String

49 *char* & '(' & *spaces* & *count_string* & ')'

50 where *char* is a replicable Character, *spaces* is a String (possibly empty) comprising only space Character values, and *count_string* is a String of one or more decimal digit Character values. A Character

replication in a picture String has the same effect as (and is said to be *equivalent to*) a String comprising *n* consecutive occurrences of *char*, where *n*=Integer'Value(*count_string*).

51 An *expanded picture String* is a picture String containing no Character replications.

51.a **Discussion:** Since 'B' is not allowed after a RHS sign, there is no need for a special rule to disallow "9.99DB(2)" as an abbreviation for "9.99DBB"

NOTES

52 3 Although a sign to the left of the number can float, a sign to the right of the number is in a fixed position.

Wording Changes from Ada 95

52.a/2 {*8652/0088*} {*AI95-00153-01*} **Corrigendum:** The picture string rules for numbers were tightened.

F.3.2 Edited Output Generation

Dynamic Semantics

1 The contents of an edited output string are based on:

2 • A value, Item, of some decimal type Num,

3 • An expanded picture String Pic_String,

4 • A Boolean value, Blank_When_Zero,

5 • A Currency string,

6 • A Fill character,

7 • A Separator character, and

8 • A Radix_Mark character.

9 The combination of a True value for Blank_When_Zero and a '*' character in Pic_String is inconsistent; no edited output string is defined.

9.a/2 **Reason:** {*AI95-00114-01*} Such a Pic_String is invalid, and any attempt to use such a string will raise Picture_Error.

10 A layout error is identified in the rules below if leading non-zero digits of Item, character values of the Currency string, or a negative sign would be truncated; in such cases no edited output string is defined.

11 The edited output string has lower bound 1 and upper bound N where N = Pic_String'Length + Currency_Length_Adjustment – Radix_Adjustment, and

12 • Currency_Length_Adjustment = Currency'Length – 1 if there is some occurrence of '$' in Pic_String, and 0 otherwise.

13 • Radix_Adjustment = 1 if there is an occurrence of 'V' or 'v' in Pic_Str, and 0 otherwise.

14 {*displayed magnitude (of a decimal value)*} Let the magnitude of Item be expressed as a base-10 number $I_p \cdots I_1.F_1 \cdots F_q$, called the *displayed magnitude* of Item, where:

15 • q = Min(Max(Num'Scale, 0), n) where n is 0 if Pic_String has no radix and is otherwise the number of digit positions following radix in Pic_String, where a digit position corresponds to an occurrence of '9', a zero_suppression_char (for an all_zero_suppression_number), a currency_char (for an all_currency_number), or a sign_char (for an all_sign_number).

16 • I_p /= 0 if p>0.

17 If n < Num'Scale, then the above number is the result of rounding (away from 0 if exactly midway between values).

18 If Blank_When_Zero = True and the displayed magnitude of Item is zero, then the edited output string comprises all space character values. Otherwise, the picture String is treated as a sequence of instances of

syntactic categories based on the rules in F.3.1, and the edited output string is the concatenation of string values derived from these categories according to the following mapping rules.

19 Table F-1 shows the mapping from a sign control symbol to a corresponding character or string in the edited output. In the columns showing the edited output, a lower-case 'b' represents the space character. If there is no sign control symbol but the value of Item is negative, a layout error occurs and no edited output string is produced.

Table F-1: Edited Output for Sign Control Symbols

Sign Control Symbol	Edited Output for Non-Negative Number	Edited Output for Negative Number
'+'	'+'	'–'
'–'	'b'	'–'
'<'	'b'	'('
'>'	'b'	')'
"CR"	"bb"	"CR"
"DB"	"bb"	"DB"

20 An instance of fixed_LHS_sign maps to a character as shown in Table F-1.

21 An instance of fixed_$_char maps to Currency.

22 An instance of direct_insertion maps to Separator if direct_insertion = '_', and to the direct_insertion Character otherwise.

23 An instance of number maps to a string *integer_part* & *radix_part* & *fraction_part* where:

- 24 The string for *integer_part* is obtained as follows:
  1. 25 Occurrences of '9' in fore_digits of number are replaced from right to left with the decimal digit character values for I_1, ..., I_p, respectively.
  2. 26 Each occurrence of '9' in fore_digits to the left of the leftmost '9' replaced according to rule 1 is replaced with '0'.
  3. 27 If p exceeds the number of occurrences of '9' in fore_digits of number, then the excess leftmost digits are eligible for use in the mapping of an instance of zero_suppression, floating_LHS_sign, floating_$_currency, or floating_#_currency to the left of number; if there is no such instance, then a layout error occurs and no edited output string is produced.
- 28 The *radix_part* is:
  - 29 "" if number does not include a radix, if radix = 'V', or if radix = 'v'
  - 30 Radix_Mark if number includes '.' as radix
- 31 The string for *fraction_part* is obtained as follows:
  1. 32 Occurrences of '9' in aft_digits of number are replaced from left to right with the decimal digit character values for F_1, ... F_q.
  2. 33 Each occurrence of '9' in aft_digits to the right of the rightmost '9' replaced according to rule 1 is replaced by '0'.

34 An instance of zero_suppression maps to the string obtained as follows:

35 1. The rightmost 'Z', 'z', or '*' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the zero_suppression instance,

36 2. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of some 'Z', 'z', or '*' in zero_suppression that has been mapped to an excess digit,

37 3. Each Character to the left of the leftmost Character replaced according to rule 1 above is replaced by:

38 - the space character if the zero suppression Character is 'Z' or 'z', or

39 - the Fill character if the zero suppression Character is '*'.

40 4. A layout error occurs if some excess digits remain after all 'Z', 'z', and '*' Character values in zero_suppression have been replaced via rule 1; no edited output string is produced.

41 An instance of RHS_sign maps to a character or string as shown in Table F-1.

42 An instance of floating_LHS_sign maps to the string obtained as follows.

43 1. Up to all but one of the rightmost LHS_Sign Character values are replaced by the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_LHS_sign instance.

44 2. The next Character to the left is replaced with the character given by the entry in Table F-1 corresponding to the LHS_Sign Character.

45 3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost LHS_Sign character replaced according to rule 1.

46 4. Any other Character is replaced by the space character..

47 5. A layout error occurs if some excess digits remain after replacement via rule 1; no edited output string is produced.

48 An instance of fixed_#_currency maps to the Currency string with n space character values concatenated on the left (if the instance does not follow a radix) or on the right (if the instance does follow a radix), where n is the difference between the length of the fixed_#_currency instance and Currency'Length. A layout error occurs if Currency'Length exceeds the length of the fixed_#_currency instance; no edited output string is produced.

49 An instance of floating_$_currency maps to the string obtained as follows:

50 1. Up to all but one of the rightmost '$' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_$_currency instance.

51 2. The next Character to the left is replaced by the Currency string.

52 3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost '$' Character replaced via rule 1.

53 4. Each other Character is replaced by the space character.

54 5. A layout error occurs if some excess digits remain after replacement by rule 1; no edited output string is produced.

55 An instance of floating_#_currency maps to the string obtained as follows:

56 1. Up to all but one of the rightmost '#' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_#_currency instance.

2. The substring whose last Character occurs at the position immediately preceding the leftmost Character replaced via rule 1, and whose length is Currency'Length, is replaced by the Currency string. 57

3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost '#' replaced via rule 1. 58

4. Any other Character is replaced by the space character. 59

5. A layout error occurs if some excess digits remain after replacement rule 1, or if there is no substring with the required length for replacement rule 2; no edited output string is produced. 60

An instance of all_zero_suppression_number maps to: 61

- a string of all spaces if the displayed magnitude of Item is zero, the zero_suppression_char is 'Z' or 'z', and the instance of all_zero_suppression_number does not have a radix at its last character position; 62
- a string containing the Fill character in each position except for the character (if any) corresponding to radix, if zero_suppression_char = '*' and the displayed magnitude of Item is zero; 63
- otherwise, the same result as if each zero_suppression_char in all_zero_suppression_aft were '9', interpreting the instance of all_zero_suppression_number as either zero_suppression number (if a radix and all_zero_suppression_aft are present), or as zero_suppression otherwise. 64

An instance of all_sign_number maps to: 65

- a string of all spaces if the displayed magnitude of Item is zero and the instance of all_sign_number does not have a radix at its last character position; 66
- otherwise, the same result as if each sign_char in all_sign_number_aft were '9', interpreting the instance of all_sign_number as either floating_LHS_sign number (if a radix and all_sign_number_aft are present), or as floating_LHS_sign otherwise. 67

An instance of all_currency_number maps to: 68

- a string of all spaces if the displayed magnitude of Item is zero and the instance of all_currency_number does not have a radix at its last character position; 69
- otherwise, the same result as if each currency_char in all_currency_number_aft were '9', interpreting the instance of all_currency_number as floating_$_currency number or floating_#_currency number (if a radix and all_currency_number_aft are present), or as floating_$_currency or floating_#_currency otherwise. 70

Examples

In the result string values shown below, 'b' represents the space character. 71

```
Item:         Picture and Result Strings:                                72
```

```
123456.78     Picture:  "-###**_***_**9.99"                              73
                        "bbb$***123,456.78"
                        "bbFF***123.456,78" (currency = "FF",
                                             separator = '.',
                                             radix mark = ',')
```

```
{8652/0089} {AI95-00070} 123456.78     Picture:  "-$**_***_**9.99"     74/1
              Result:   "b$***123,456.78"
                       "bFF***123.456,78" (currency = "FF",
                                           separator = '.',
                                           radix mark = ',')
```

```
0.0          Picture: "-$$$$$$.$$"                                       75
             Result:  "bbbbbbbbbb"
```

76
```
0.20          Picture: "-$$$$$$.$$"
              Result:  "bbbbbb$.20"
```

77
```
-1234.565     Picture: "<<<<_<<<.<<###>"
              Result:  "bb(1,234.57DMb)"  (currency = "DM")
```

78
```
12345.67      Picture: "###_###_##9.99"
              Result:  "bbCHF12,345.67"   (currency = "CHF")
```

Wording Changes from Ada 95

78.a/2 {*8652/0089*} {*AI95-00070-01*} **Corrigendum:** Corrected the picture string example.

F.3.3 The Package Text_IO.Editing

1 The package Text_IO.Editing provides a private type Picture with associated operations, and a generic package Decimal_Output. An object of type Picture is composed from a well-formed picture String (see F.3.1) and a Boolean item indicating whether a zero numeric value will result in an edited output string of all space characters. The package Decimal_Output contains edited output subprograms implementing the effects defined in F.3.2.

Static Semantics

2 The library package Text_IO.Editing has the following declaration:

3
```
package Ada.Text_IO.Editing is
```
4
```
   type Picture is private;
```
5
```
   function Valid (Pic_String      : in String;
                   Blank_When_Zero : in Boolean := False) return Boolean;
```
6
```
   function To_Picture (Pic_String      : in String;
                        Blank_When_Zero : in Boolean := False)
      return Picture;
```
7
```
   function Pic_String      (Pic : in Picture) return String;
   function Blank_When_Zero (Pic : in Picture) return Boolean;
```
8
```
   Max_Picture_Length  : constant := implementation_defined;
```
9
```
   Picture_Error       : exception;
```
10
```
   Default_Currency    : constant String    := "$";
   Default_Fill        : constant Character := '*';
   Default_Separator   : constant Character := ',';
   Default_Radix_Mark  : constant Character := '.';
```
11
```
   generic
      type Num is delta <> digits <>;
      Default_Currency   : in String    := Text_IO.Editing.Default_Currency;
      Default_Fill       : in Character := Text_IO.Editing.Default_Fill;
      Default_Separator  : in Character :=
                              Text_IO.Editing.Default_Separator;
      Default_Radix_Mark : in Character :=
                              Text_IO.Editing.Default_Radix_Mark;
   package Decimal_Output is
      function Length (Pic      : in Picture;
                       Currency : in String := Default_Currency)
         return Natural;
```
12
```
      function Valid (Item     : in Num;
                      Pic      : in Picture;
                      Currency : in String := Default_Currency)
         return Boolean;
```

13
```
      function Image (Item       : in Num;
                      Pic        : in Picture;
                      Currency   : in String    := Default_Currency;
                      Fill       : in Character := Default_Fill;
                      Separator  : in Character := Default_Separator;
                      Radix_Mark : in Character := Default_Radix_Mark)
         return String;
```

14
```
      procedure Put (File       : in File_Type;
                     Item       : in Num;
                     Pic        : in Picture;
                     Currency   : in String    := Default_Currency;
                     Fill       : in Character := Default_Fill;
                     Separator  : in Character := Default_Separator;
                     Radix_Mark : in Character := Default_Radix_Mark);
```

15
```
      procedure Put (Item       : in Num;
                     Pic        : in Picture;
                     Currency   : in String    := Default_Currency;
                     Fill       : in Character := Default_Fill;
                     Separator  : in Character := Default_Separator;
                     Radix_Mark : in Character := Default_Radix_Mark);
```

16
```
      procedure Put (To         : out String;
                     Item       : in Num;
                     Pic        : in Picture;
                     Currency   : in String    := Default_Currency;
                     Fill       : in Character := Default_Fill;
                     Separator  : in Character := Default_Separator;
                     Radix_Mark : in Character := Default_Radix_Mark);
   end Decimal_Output;
private
   ... -- not specified by the language
end Ada.Text_IO.Editing;
```

16.a **Implementation defined:** The value of Max_Picture_Length in the package Text_IO.Editing

17 The exception Constraint_Error is raised if the Image function or any of the Put procedures is invoked with a null string for Currency.

18
```
function Valid (Pic_String      : in String;
                Blank_When_Zero : in Boolean := False) return Boolean;
```

19 Valid returns True if Pic_String is a well-formed picture String (see F.3.1) the length of whose expansion does not exceed Max_Picture_Length, and if either Blank_When_Zero is False or Pic_String contains no '*'.

20
```
function To_Picture (Pic_String      : in String;
                     Blank_When_Zero : in Boolean := False)
   return Picture;
```

21 To_Picture returns a result Picture such that the application of the function Pic_String to this result yields an expanded picture String equivalent to Pic_String, and such that Blank_When_Zero applied to the result Picture is the same value as the parameter Blank_When_Zero. Picture_Error is raised if not Valid(Pic_String, Blank_When_Zero).

22
```
function Pic_String      (Pic : in Picture) return String;

function Blank_When_Zero (Pic : in Picture) return Boolean;
```

23 If Pic is To_Picture(String_Item, Boolean_Item) for some String_Item and Boolean_Item, then:

- 24 Pic_String(Pic) returns an expanded picture String equivalent to String_Item and with any lower-case letter replaced with its corresponding upper-case form, and
- 25 Blank_When_Zero(Pic) returns Boolean_Item.

26 If Pic_1 and Pic_2 are objects of type Picture, then "="(Pic_1, Pic_2) is True when

27
- Pic_String(Pic_1) = Pic_String(Pic_2), and

28
- Blank_When_Zero(Pic_1) = Blank_When_Zero(Pic_2).

29
```
function Length (Pic      : in Picture;
                 Currency : in String := Default_Currency)
   return Natural;
```

30 Length returns Pic_String(Pic)'Length + Currency_Length_Adjustment – Radix_Adjustment where

31
- Currency_Length_Adjustment =

32
  - Currency'Length – 1 if there is some occurrence of '$' in Pic_String(Pic), and

33
  - 0 otherwise.

34
- Radix_Adjustment =

35
  - 1 if there is an occurrence of 'V' or 'v' in Pic_Str(Pic), and

36
  - 0 otherwise.

37
```
function Valid (Item     : in Num;
                Pic      : in Picture;
                Currency : in String := Default_Currency)
   return Boolean;
```

38 Valid returns True if Image(Item, Pic, Currency) does not raise Layout_Error, and returns False otherwise.

39
```
function Image (Item       : in Num;
                Pic        : in Picture;
                Currency   : in String    := Default_Currency;
                Fill       : in Character := Default_Fill;
                Separator  : in Character := Default_Separator;
                Radix_Mark : in Character := Default_Radix_Mark)
   return String;
```

40 Image returns the edited output String as defined in F.3.2 for Item, Pic_String(Pic), Blank_When_Zero(Pic), Currency, Fill, Separator, and Radix_Mark. If these rules identify a layout error, then Image raises the exception Layout_Error.

41
```
procedure Put (File       : in File_Type;
               Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);

procedure Put (Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);
```

42 Each of these Put procedures outputs Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) consistent with the conventions for Put for other real types in case of bounded line length (see A.10.6, “Get and Put Procedures”).

```
procedure Put (To         : out String;                                43
               Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);
```

Put copies Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) to the given string, right justified. Otherwise unassigned Character values in To are assigned the space character. If To'Length is less than the length of the string resulting from Image, then Layout_Error is raised. 44

Implementation Requirements

Max_Picture_Length shall be at least 30. The implementation shall support currency strings of length up to at least 10, both for Default_Currency in an instantiation of Decimal_Output, and for Currency in an invocation of Image or any of the Put procedures. 45

Discussion: This implies that a picture string with character replications need not be supported (i.e., To_Picture will raise Picture_Error) if its expanded form exceeds 30 characters. 45.a

NOTES

4 The rules for edited output are based on COBOL (ANSI X3.23:1985, endorsed by ISO as ISO 1989-1985), with the following differences: 46

- The COBOL provisions for picture string localization and for 'P' format are absent from Ada. 47
- The following Ada facilities are not in COBOL: 48
  - currency symbol placement after the number, 49
  - localization of edited output string for multi-character currency string values, including support for both length-preserving and length-expanding currency symbols in picture strings 50
  - localization of the radix mark, digits separator, and fill character, and 51
  - parenthesization of negative values. 52

The value of 30 for Max_Picture_Length is the same limit as in COBOL. 52.1

Reason: There are several reasons we have not adopted the COBOL-style permission to provide a single-character replacement in the picture string for the `$' as currency symbol, or to interchange the roles of `.' and `,' in picture strings 52.a

- It would have introduced considerable complexity into Ada, as well as confusion between run-time and compile-time character interpretation, since picture Strings are dynamically computable in Ada, in contrast with COBOL 52.b
- Ada's rules for real literals provide a natural interpretation of `_' as digits separator and `.' for radix mark; it is not essential to allow these to be localized in picture strings, since Ada does not allow them to be localized in real literals. 52.c
- The COBOL restriction for the currency symbol in a picture string to be replaced by a single character currency symbol is a compromise solution. For general international usage a mechanism is needed to localize the edited output to be a multi-character currency string. Allowing a single-Character localization for the picture Character, and a multiple-character localization for the currency string, would be an unnecessary complication. 52.d

F.3.4 The Package Wide_Text_IO.Editing

Static Semantics

{*Ada.Wide_*} The child package Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that: 1

- each occurrence of Character is replaced by Wide_Character, 2
- each occurrence of Text_IO is replaced by Wide_Text_IO, 3
- the subtype of Default_Currency is Wide_String rather than String, and 4
- each occurrence of String in the generic package Decimal_Output is replaced by Wide_String. 5

5.a **Implementation defined:** The value of Max_Picture_Length in the package Wide_Text_IO.Editing

NOTES

6 5 Each of the functions Wide_Text_IO.Editing.Valid, To_Picture, and Pic_String has String (versus Wide_String) as its parameter or result subtype, since a picture String is not localizable.

F.3.5 The Package Wide_Wide_Text_IO.Editing

Static Semantics

1/2 {*AI95-00285-01*} {*Ada.Wide_*} The child package Wide_Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that:

- 2/2 each occurrence of Character is replaced by Wide_Wide_Character,
- 3/2 each occurrence of Text_IO is replaced by Wide_Wide_Text_IO,
- 4/2 the subtype of Default_Currency is Wide_Wide_String rather than String, and
- 5/2 each occurrence of String in the generic package Decimal_Output is replaced by Wide_Wide_String.

5.a/2 **Implementation defined:** The value of Max_Picture_Length in the package Wide_Wide_Text_IO.Editing

NOTES

6/2 6 {*AI95-00285-01*} Each of the functions Wide_Wide_Text_IO.Editing.Valid, To_Picture, and Pic_String has String (versus Wide_Wide_String) as its parameter or result subtype, since a picture String is not localizable.

Extensions to Ada 95

6.a/2 {*AI95-00285-01*} {*extensions to Ada 95*} Package Wide_Wide_Text_IO.Editing is new; it supports 32-bit character strings. (Shouldn't it have been “Widest_Text_IO.Editing”? :-)

Annex G (normative) Numerics

{*numerics*} The Numerics Annex specifies 1

- features for complex arithmetic, including complex I/O; 2
- a mode ("strict mode"), in which the predefined arithmetic operations of floating point and fixed point types and the functions and operations of various predefined packages have to provide guaranteed accuracy or conform to other numeric performance requirements, which the Numerics Annex also specifies; 3
- a mode ("relaxed mode"), in which no accuracy or other numeric performance requirements need be satisfied, as for implementations not conforming to the Numerics Annex; 4
- {*AI95-00296-01*} models of floating point and fixed point arithmetic on which the accuracy requirements of strict mode are based; 5/2
- {*AI95-00296-01*} the definitions of the model-oriented attributes of floating point types that apply in the strict mode; and 6/2
- {*AI95-00296-01*} features for the manipulation of real and complex vectors and matrices. 6.1/2

Implementation Advice

If Fortran (respectively, C) is widely supported in the target environment, implementations supporting the Numerics Annex should provide the child package Interfaces.Fortran (respectively, Interfaces.C) specified in Annex B and should support a *convention_*identifier of Fortran (respectively, C) in the interfacing pragmas (see Annex B), thus allowing Ada programs to interface with programs written in that language. 7

Implementation Advice: If Fortran (respectively, C) is supported in the target environment, then interfacing to Fortran (respectively, C) should be supported as specified in Annex B. 7.a.1/2

Extensions to Ada 83

{*extensions to Ada 83*} This Annex is new to Ada 95. 7.a

G.1 Complex Arithmetic

Types and arithmetic operations for complex arithmetic are provided in Generic_Complex_Types, which is defined in G.1.1. Implementation-defined approximations to the complex analogs of the mathematical functions known as the "elementary functions" are provided by the subprograms in Generic_Complex_-Elementary_Functions, which is defined in G.1.2. Both of these library units are generic children of the predefined package Numerics (see A.5). Nongeneric equivalents of these generic packages for each of the predefined floating point types are also provided as children of Numerics. 1

Implementation defined: The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations. 1.a

Discussion: Complex arithmetic is defined in the Numerics Annex, rather than in the core, because it is considered to be a specialized need of (some) numeric applications. 1.b

G.1.1 Complex Types

Static Semantics

1 The generic library package Numerics.Generic_Complex_Types has the following declaration:

2/1
```
{8652/0020} {AI95-00126-01} generic
   type Real is digits <>;
package Ada.Numerics.Generic_Complex_Types is
   pragma Pure(Generic_Complex_Types);
```

3
```
   type Complex is
      record
         Re, Im : Real'Base;
      end record;
```

4/2
```
{AI95-00161-01}    type Imaginary is private;
   pragma Preelaborable_Initialization(Imaginary);
```

5
```
   i : constant Imaginary;
   j : constant Imaginary;
```

6
```
   function Re (X : Complex)   return Real'Base;
   function Im (X : Complex)   return Real'Base;
   function Im (X : Imaginary) return Real'Base;
```

7
```
   procedure Set_Re (X  : in out Complex;
                     Re : in     Real'Base);
   procedure Set_Im (X  : in out Complex;
                     Im : in     Real'Base);
   procedure Set_Im (X  :    out Imaginary;
                     Im : in     Real'Base);
```

8
```
   function Compose_From_Cartesian (Re, Im : Real'Base) return Complex;
   function Compose_From_Cartesian (Re     : Real'Base) return Complex;
   function Compose_From_Cartesian (Im     : Imaginary) return Complex;
```

9
```
   function Modulus (X     : Complex) return Real'Base;
   function "abs"   (Right : Complex) return Real'Base renames Modulus;
```

10
```
   function Argument (X     : Complex)   return Real'Base;
   function Argument (X     : Complex;
                      Cycle : Real'Base) return Real'Base;
```

11
```
   function Compose_From_Polar (Modulus, Argument        : Real'Base)
      return Complex;
   function Compose_From_Polar (Modulus, Argument, Cycle : Real'Base)
      return Complex;
```

12
```
   function "+"       (Right : Complex) return Complex;
   function "-"       (Right : Complex) return Complex;
   function Conjugate (X     : Complex) return Complex;
```

13
```
   function "+" (Left, Right : Complex) return Complex;
   function "-" (Left, Right : Complex) return Complex;
   function "*" (Left, Right : Complex) return Complex;
   function "/" (Left, Right : Complex) return Complex;
```

14
```
   function "**" (Left : Complex; Right : Integer) return Complex;
```

15
```
   function "+"       (Right : Imaginary) return Imaginary;
   function "-"       (Right : Imaginary) return Imaginary;
   function Conjugate (X     : Imaginary) return Imaginary renames "-";
   function "abs"     (Right : Imaginary) return Real'Base;
```

16
```
   function "+" (Left, Right : Imaginary) return Imaginary;
   function "-" (Left, Right : Imaginary) return Imaginary;
   function "*" (Left, Right : Imaginary) return Real'Base;
   function "/" (Left, Right : Imaginary) return Real'Base;
```

17
```
   function "**" (Left : Imaginary; Right : Integer) return Complex;
```

```
18      function "<"  (Left, Right : Imaginary) return Boolean;
        function "<=" (Left, Right : Imaginary) return Boolean;
        function ">"  (Left, Right : Imaginary) return Boolean;
        function ">=" (Left, Right : Imaginary) return Boolean;

19      function "+" (Left : Complex;   Right : Real'Base) return Complex;
        function "+" (Left : Real'Base; Right : Complex)   return Complex;
        function "-" (Left : Complex;   Right : Real'Base) return Complex;
        function "-" (Left : Real'Base; Right : Complex)   return Complex;
        function "*" (Left : Complex;   Right : Real'Base) return Complex;
        function "*" (Left : Real'Base; Right : Complex)   return Complex;
        function "/" (Left : Complex;   Right : Real'Base) return Complex;
        function "/" (Left : Real'Base; Right : Complex)   return Complex;

20      function "+" (Left : Complex;   Right : Imaginary) return Complex;
        function "+" (Left : Imaginary; Right : Complex)   return Complex;
        function "-" (Left : Complex;   Right : Imaginary) return Complex;
        function "-" (Left : Imaginary; Right : Complex)   return Complex;
        function "*" (Left : Complex;   Right : Imaginary) return Complex;
        function "*" (Left : Imaginary; Right : Complex)   return Complex;
        function "/" (Left : Complex;   Right : Imaginary) return Complex;
        function "/" (Left : Imaginary; Right : Complex)   return Complex;

21      function "+" (Left : Imaginary; Right : Real'Base) return Complex;
        function "+" (Left : Real'Base; Right : Imaginary) return Complex;
        function "-" (Left : Imaginary; Right : Real'Base) return Complex;
        function "-" (Left : Real'Base; Right : Imaginary) return Complex;
        function "*" (Left : Imaginary; Right : Real'Base) return Imaginary;
        function "*" (Left : Real'Base; Right : Imaginary) return Imaginary;
        function "/" (Left : Imaginary; Right : Real'Base) return Imaginary;
        function "/" (Left : Real'Base; Right : Imaginary) return Imaginary;

22   private

23      type Imaginary is new Real'Base;
        i : constant Imaginary := 1.0;
        j : constant Imaginary := 1.0;

24   end Ada.Numerics.Generic_Complex_Types;
```

25/1 {*8652/0020*} {*AI95-00126-01*} The library package Numerics.Complex_Types is declared pure and defines the same types, constants, and subprograms as Numerics.Generic_Complex_Types, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents of Numerics.Generic_Complex_Types for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Types, Numerics.Long_Complex_Types, etc.

25.a **Reason:** The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics.

25.b **Reason:** The nongeneric equivalents all export the types Complex and Imaginary and the constants i and j (rather than uniquely named types and constants, such as Short_Complex, Long_Complex, etc.) to preserve their equivalence to actual instantiations of the generic package and to allow the programmer to change the precision of an application globally by changing a single context clause.

26/2 {*AI95-00434-01*} [Complex is a visible type with Cartesian components.]

26.a **Reason:** The Cartesian representation is far more common than the polar representation, in practice. The accuracy of the results of the complex arithmetic operations and of the complex elementary functions is dependent on the representation; thus, implementers need to know that representation. The type is visible so that complex “literals” can be written in aggregate notation, if desired.

27 [Imaginary is a private type; its full type is derived from Real'Base.]

27.a **Reason:** The Imaginary type and the constants i and j are provided for two reasons:

- 27.b They allow complex “literals” to be written in the alternate form of $a + b$*i (or $a + b$*j), if desired. Of course, in some contexts the sum will need to be parenthesized.
- 27.c When an Ada binding to IEC 559:1989 that provides (signed) infinities as the result of operations that overflow becomes available, it will be important to allow arithmetic between pure-imaginary and complex operands without requiring the former to be represented as (or promoted to) complex values with a real component of zero. For example, the multiplication of $a + b$*i by d*i should yield $-b\cdot d + a\cdot d$*i, but if one

cannot avoid representing the pure-imaginary value *d**i as the complex value 0.0 + *d**i, then a NaN ("Not-a-Number") could be produced as the result of multiplying *a* by 0.0 (e.g., when *a* is infinite); the NaN could later trigger an exception. Providing the Imaginary type and overloadings of the arithmetic operators for mixtures of Imaginary and Complex operands gives the programmer the same control over avoiding premature coercion of pure-imaginary values to complex as is already provided for pure-real values.

27.d **Reason:** The Imaginary type is private, rather than being visibly derived from Real'Base, for two reasons:

27.e • to preclude implicit conversions of real literals to the Imaginary type (such implicit conversions would make many common arithmetic expressions ambiguous); and

27.f • to suppress the implicit derivation of the multiplication, division, and absolute value operators with Imaginary operands and an Imaginary result (the result type would be incorrect).

27.g **Reason:** The base subtype Real'Base is used for the component type of Complex, the parent type of Imaginary, and the parameter and result types of some of the subprograms to maximize the chances of being able to pass meaningful values into the subprograms and receive meaningful results back. The generic formal parameter Real therefore plays only one role, that of providing the precision to be maintained in complex arithmetic calculations. Thus, the subprograms in Numerics.Generic_Complex_Types share with those in Numerics.Generic_Elementary_Functions, and indeed even with the predefined arithmetic operations (see 4.5), the property of being free of range checks on input and output, i.e., of being able to exploit the base range of the relevant floating point type fully. As a result, the user loses the ability to impose application-oriented bounds on the range of values that the components of a complex variable can acquire; however, it can be argued that few, if any, applications have a naturally square domain (as opposed to a circular domain) anyway.

28 The arithmetic operations and the Re, Im, Modulus, Argument, and Conjugate functions have their usual mathematical meanings. When applied to a parameter of pure-imaginary type, the "imaginary-part" function Im yields the value of its parameter, as the corresponding real value. The remaining subprograms have the following meanings:

28.a **Reason:** The middle case can be understood by considering the parameter of pure-imaginary type to represent a complex value with a zero real part.

29 • The Set_Re and Set_Im procedures replace the designated component of a complex parameter with the given real value; applied to a parameter of pure-imaginary type, the Set_Im procedure replaces the value of that parameter with the imaginary value corresponding to the given real value.

30 • The Compose_From_Cartesian function constructs a complex value from the given real and imaginary components. If only one component is given, the other component is implicitly zero.

31 • The Compose_From_Polar function constructs a complex value from the given modulus (radius) and argument (angle). When the value of the parameter Modulus is positive (resp., negative), the result is the complex value represented by the point in the complex plane lying at a distance from the origin given by the absolute value of Modulus and forming an angle measured counterclockwise from the positive (resp., negative) real axis given by the value of the parameter Argument.

32 When the Cycle parameter is specified, the result of the Argument function and the parameter Argument of the Compose_From_Polar function are measured in units such that a full cycle of revolution has the given value; otherwise, they are measured in radians.

33 The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

34 • The result of the Modulus function is nonnegative.

35 • The result of the Argument function is in the quadrant containing the point in the complex plane represented by the parameter X. This may be any quadrant (I through IV); thus, the range of the Argument function is approximately $-\pi$ to π (–Cycle/2.0 to Cycle/2.0, if the parameter Cycle is specified). When the point represented by the parameter X lies on the negative real axis, the result approximates

36 • π (resp., $-\pi$) when the sign of the imaginary component of X is positive (resp., negative), if Real'Signed_Zeros is True;

  - π, if Real'Signed_Zeros is False. 37

- Because a result lying on or near one of the axes may not be exactly representable, the approximation inherent in computing the result may place it in an adjacent quadrant, close to but on the wrong side of the axis. 38

Dynamic Semantics

39 The exception Numerics.Argument_Error is raised by the Argument and Compose_From_Polar functions with specified cycle, signaling a parameter value outside the domain of the corresponding mathematical function, when the value of the parameter Cycle is zero or negative.

40 {*Division_Check* [partial]} {*check, language-defined (Division_Check)*} {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised by the division operator when the value of the right operand is zero, and by the exponentiation operator when the value of the left operand is zero and the value of the exponent is negative, provided that Real'Machine_Overflows is True; when Real'Machine_Overflows is False, the result is unspecified. {*unspecified* [partial]} [Constraint_Error can also be raised when a finite result overflows (see G.2.6).]

40.a **Discussion:** It is anticipated that an Ada binding to IEC 559:1989 will be developed in the future. As part of such a binding, the Machine_Overflows attribute of a conformant floating point type will be specified to yield False, which will permit implementations of the complex arithmetic operations to deliver results with an infinite component (and set the overflow flag defined by the binding) instead of raising Constraint_Error in overflow situations, when traps are disabled. Similarly, it is appropriate for the complex arithmetic operations to deliver results with infinite components (and set the zero-divide flag defined by the binding) instead of raising Constraint_Error in the situations defined above, when traps are disabled. Finally, such a binding should also specify the behavior of the complex arithmetic operations, when sensible, given operands with infinite components.

Implementation Requirements

41 In the implementation of Numerics.Generic_Complex_Types, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Real.

41.a **Implementation Note:** Implementations of Numerics.Generic_Complex_Types written in Ada should therefore avoid declaring local variables of subtype Real; the subtype Real'Base should be used instead.

42 {*prescribed result (for the evaluation of a complex arithmetic operation)*} In the following cases, evaluation of a complex arithmetic operation shall yield the *prescribed result*, provided that the preceding rules do not call for an exception to be raised:

- The results of the Re, Im, and Compose_From_Cartesian functions are exact. 43
- The real (resp., imaginary) component of the result of a binary addition operator that yields a result of complex type is exact when either of its operands is of pure-imaginary (resp., real) type. 44

44.a **Ramification:** The result of the addition operator is exact when one of its operands is of real type and the other is of pure-imaginary type. In this particular case, the operator is analogous to the Compose_From_Cartesian function; it performs no arithmetic.

- The real (resp., imaginary) component of the result of a binary subtraction operator that yields a result of complex type is exact when its right operand is of pure-imaginary (resp., real) type. 45
- The real component of the result of the Conjugate function for the complex type is exact. 46
- When the point in the complex plane represented by the parameter X lies on the nonnegative real axis, the Argument function yields a result of zero. 47

47.a **Discussion:** Argument(X + i*Y) is analogous to *EF*.Arctan(Y, X), where *EF* is an appropriate instance of Numerics.Generic_Elementary_Functions, except when X and Y are both zero, in which case the former yields the value zero while the latter raises Numerics.Argument_Error.

- When the value of the parameter Modulus is zero, the Compose_From_Polar function yields a result of zero. 48

49 • When the value of the parameter Argument is equal to a multiple of the quarter cycle, the result of the Compose_From_Polar function with specified cycle lies on one of the axes. In this case, one of its components is zero, and the other has the magnitude of the parameter Modulus.

50 • Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand. Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero, provided that the exponent is nonzero. When the left operand is of pure-imaginary type, one component of the result of the exponentiation operator is zero.

51 When the result, or a result component, of any operator of Numerics.Generic_Complex_Types has a mathematical definition in terms of a single arithmetic or relational operation, that result or result component exhibits the accuracy of the corresponding operation of the type Real.

52 Other accuracy requirements for the Modulus, Argument, and Compose_From_Polar functions, and accuracy requirements for the multiplication of a pair of complex operands or for division by a complex operand, all of which apply only in the strict mode, are given in G.2.6.

53 The sign of a zero result or zero result component yielded by a complex arithmetic operation or function is implementation defined when Real'Signed_Zeros is True.

53.a **Implementation defined:** The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Types, when Real'Signed_Zeros is True.

Implementation Permissions

54 The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

55/2 {*8652/0091*} {*AI95-00434-01*} Implementations may obtain the result of exponentiation of a complex or pure-imaginary operand by repeated complex multiplication, with arbitrary association of the factors and with a possible final complex reciprocation (when the exponent is negative). Implementations are also permitted to obtain the result of exponentiation of a complex operand, but not of a pure-imaginary operand, by converting the left operand to a polar representation; exponentiating the modulus by the given exponent; multiplying the argument by the given exponent; and reconverting to a Cartesian representation. Because of this implementation freedom, no accuracy requirement is imposed on complex exponentiation (except for the prescribed results given above, which apply regardless of the implementation method chosen).

Implementation Advice

56 Because the usual mathematical meaning of multiplication of a complex operand and a real operand is that of the scaling of both components of the former by the latter, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex multiplication. In systems that, in the future, support an Ada binding to IEC 559:1989, the latter technique will not generate the required result when one of the components of the complex operand is infinite. (Explicit multiplication of the infinite component by the zero component obtained during promotion yields a NaN that propagates into the final result.) Analogous advice applies in the case of multiplication of a complex operand and a pure-imaginary operand, and in the case of division of a complex operand by a real or pure-imaginary operand.

56.a/2 **Implementation Advice:** Mixed real and complex operations (as well as pure-imaginary and complex operations) should not be performed by converting the real (resp. pure-imaginary) operand to complex.

57 Likewise, because the usual mathematical meaning of addition of a complex operand and a real operand is that the imaginary operand remains unchanged, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex addition. In

implementations in which the Signed_Zeros attribute of the component type is True (and which therefore conform to IEC 559:1989 in regard to the handling of the sign of zero in predefined arithmetic operations), the latter technique will not generate the required result when the imaginary component of the complex operand is a negatively signed zero. (Explicit addition of the negative zero to the zero obtained during promotion yields a positive zero.) Analogous advice applies in the case of addition of a complex operand and a pure-imaginary operand, and in the case of subtraction of a complex operand and a real or pure-imaginary operand.

Implementations in which Real'Signed_Zeros is True should attempt to provide a rational treatment of the signs of zero results and result components. As one example, the result of the Argument function should have the sign of the imaginary component of the parameter X when the point represented by that parameter lies on the positive real axis; as another, the sign of the imaginary component of the Compose_From_Polar function should be the same as (resp., the opposite of) that of the Argument parameter when that parameter has a value of zero and the Modulus parameter has a nonnegative (resp., negative) value. 58

Implementation Advice: If Real'Signed_Zeros is true for Numerics.Generic_Complex_Types, a rational treatment of the signs of zero results and result components should be provided. 58.a.1/2

Wording Changes from Ada 83

The semantics of Numerics.Generic_Complex_Types differs from Generic_Complex_Types as defined in ISO/IEC CD 13813 (for Ada 83) in the following ways: 58.a

- The generic package is a child of the package defining the Argument_Error exception. 58.b
- The nongeneric equivalents export types and constants with the same names as those exported by the generic package, rather than with names unique to the package. 58.c
- Implementations are not allowed to impose an optional restriction that the generic actual parameter associated with Real be unconstrained. (In view of the ability to declare variables of subtype Real'Base in implementations of Numerics.Generic_Complex_Types, this flexibility is no longer needed.) 58.d
- The dependence of the Argument function on the sign of a zero parameter component is tied to the value of Real'Signed_Zeros. 58.e
- Conformance to accuracy requirements is conditional. 58.f

Extensions to Ada 95

{*AI95-00161-01*} {*extensions to Ada 95*} **Amendment Correction:** Added a pragma Preelaborable_Initialization to type Imaginary, so that it can be used in preelaborated units. 58.g/2

Wording Changes from Ada 95

{*8652/0020*} {*AI95-00126-01*} **Corrigendum:** Explicitly stated that the nongeneric equivalents of Generic_Complex_Types are pure. 58.h/2

G.1.2 Complex Elementary Functions

Static Semantics

The generic library package Numerics.Generic_Complex_Elementary_Functions has the following declaration: 1

2/2
```
{AI95-00434-01} with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (<>);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Elementary_Functions is
   pragma Pure(Generic_Complex_Elementary_Functions);
```

3
```
      function Sqrt (X : Complex)   return Complex;
      function Log  (X : Complex)   return Complex;
      function Exp  (X : Complex)   return Complex;
      function Exp  (X : Imaginary) return Complex;
      function "**" (Left : Complex;   Right : Complex)   return Complex;
      function "**" (Left : Complex;   Right : Real'Base) return Complex;
      function "**" (Left : Real'Base; Right : Complex)   return Complex;
```
4
```
      function Sin (X : Complex) return Complex;
      function Cos (X : Complex) return Complex;
      function Tan (X : Complex) return Complex;
      function Cot (X : Complex) return Complex;
```
5
```
      function Arcsin (X : Complex) return Complex;
      function Arccos (X : Complex) return Complex;
      function Arctan (X : Complex) return Complex;
      function Arccot (X : Complex) return Complex;
```
6
```
      function Sinh (X : Complex) return Complex;
      function Cosh (X : Complex) return Complex;
      function Tanh (X : Complex) return Complex;
      function Coth (X : Complex) return Complex;
```
7
```
      function Arcsinh (X : Complex) return Complex;
      function Arccosh (X : Complex) return Complex;
      function Arctanh (X : Complex) return Complex;
      function Arccoth (X : Complex) return Complex;
```
8
```
   end Ada.Numerics.Generic_Complex_Elementary_Functions;
```

9/1 {*8652/0020*} {*AI95-00126-01*} The library package Numerics.Complex_Elementary_Functions is declared pure and defines the same subprograms as Numerics.Generic_Complex_Elementary_Functions, except that the predefined type Float is systematically substituted for Real'Base, and the Complex and Imaginary types exported by Numerics.Complex_Types are systematically substituted for Complex and Imaginary, throughout. Nongeneric equivalents of Numerics.Generic_Complex_Elementary_Functions corresponding to each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Elementary_Functions, Numerics.Long_Complex_Elementary_Functions, etc.

9.a **Reason:** The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics.

10 The overloading of the Exp function for the pure-imaginary type is provided to give the user an alternate way to compose a complex value from a given modulus and argument. In addition to Compose_From_-Polar(Rho, Theta) (see G.1.1), the programmer may write Rho * Exp(i * Theta).

11 The imaginary (resp., real) component of the parameter X of the forward hyperbolic (resp., trigonometric) functions and of the Exp function (and the parameter X, itself, in the case of the overloading of the Exp function for the pure-imaginary type) represents an angle measured in radians, as does the imaginary (resp., real) component of the result of the Log and inverse hyperbolic (resp., trigonometric) functions.

12 The functions have their usual mathematical meanings. However, the arbitrariness inherent in the placement of branch cuts, across which some of the complex elementary functions exhibit discontinuities, is eliminated by the following conventions:

13 • The imaginary component of the result of the Sqrt and Log functions is discontinuous as the parameter X crosses the negative real axis.

14 • The result of the exponentiation operator when the left operand is of complex type is discontinuous as that operand crosses the negative real axis.

15/2 • {*AI95-00185-01*} The imaginary component of the result of the Arcsin, Arccos, and Arctanh functions is discontinuous as the parameter X crosses the real axis to the left of –1.0 or the right of 1.0.

- {*AI95-00185-01*} The real component of the result of the Arctan and Arcsinh functions is discontinuous as the parameter X crosses the imaginary axis below –*i* or above *i*. 16/2
- {*AI95-00185-01*} The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis below –*i* or above *i*. 17/2
- The imaginary component of the Arccosh function is discontinuous as the parameter X crosses the real axis to the left of 1.0. 18
- The imaginary component of the result of the Arccoth function is discontinuous as the parameter X crosses the real axis between –1.0 and 1.0. 19

  Discussion: {*AI95-00185-01*} The branch cuts come from the fact that the functions in question are really multi-valued in the complex domain, and that we have to pick one *principal value* to be the result of the function. Evidently we have much freedom in choosing where the branch cuts lie. However, we are adhering to the following principles which seem to lead to the more *natural* definitions: 19.a/2

  - A branch cut should not intersect the real axis at a place where the corresponding real function is well-defined (in other words, the complex function should be an extension of the corresponding real function). 19.b/2
  - Because all the functions in question are analytic, to ensure power series validity for the principal value, the branch cuts should be invariant by complex conjugation. 19.c/2
  - For odd functions, to ensure that the principal value remains an odd function, the branch cuts should be invariant by reflection in the origin. 19.d/2

{*AI95-00185-01*} The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply that the principal branch is an analytic continuation of the corresponding real-valued function in Numerics.Generic_Elementary_Functions. (For Arctan and Arccot, the single-argument function in question is that obtained from the two-argument version by fixing the second argument to be its default value.) 20/2

- The real component of the result of the Sqrt and Arccosh functions is nonnegative. 21
- The same convention applies to the imaginary component of the result of the Log function as applies to the result of the natural-cycle version of the Argument function of Numerics.Generic_Complex_Types (see G.1.1). 22
- The range of the real (resp., imaginary) component of the result of the Arcsin and Arctan (resp., Arcsinh and Arctanh) functions is approximately –π/2.0 to π/2.0. 23
- The real (resp., imaginary) component of the result of the Arccos and Arccot (resp., Arccoth) functions ranges from 0.0 to approximately π. 24
- The range of the imaginary component of the result of the Arccosh function is approximately –π to π. 25

In addition, the exponentiation operator inherits the single-valuedness of the Log function. 26

Dynamic Semantics

The exception Numerics.Argument_Error is raised by the exponentiation operator, signaling a parameter value outside the domain of the corresponding mathematical function, when the value of the left operand is zero and the real component of the exponent (or the exponent itself, when it is of real type) is zero. 27

{*Division_Check* [partial]} {*check, language-defined (Division_Check)*} {*Constraint_Error (raised by failure of run-time check)*} The exception Constraint_Error is raised, signaling a pole of the mathematical function (analogous to dividing by zero), in the following cases, provided that Complex_Types.Real'Machine_Overflows is True: 28

- by the Log, Cot, and Coth functions, when the value of the parameter X is zero; 29
- by the exponentiation operator, when the value of the left operand is zero and the real component of the exponent (or the exponent itself, when it is of real type) is negative; 30

31 • by the Arctan and Arccot functions, when the value of the parameter X is ± *i*;

32 • by the Arctanh and Arccoth functions, when the value of the parameter X is ± 1.0.

33 [Constraint_Error can also be raised when a finite result overflows (see G.2.6); this may occur for parameter values sufficiently *near* poles, and, in the case of some of the functions, for parameter values having components of sufficiently large magnitude.] {*unspecified* [partial]} When Complex_Types.Real'Machine_Overflows is False, the result at poles is unspecified.

33.a **Reason:** The purpose of raising Constraint_Error (rather than Numerics.Argument_Error) at the poles of a function, when Float_Type'Machine_Overflows is True, is to provide continuous behavior as the actual parameters of the function approach the pole and finally reach it.

33.b **Discussion:** It is anticipated that an Ada binding to IEC 559:1989 will be developed in the future. As part of such a binding, the Machine_Overflows attribute of a conformant floating point type will be specified to yield False, which will permit implementations of the complex elementary functions to deliver results with an infinite component (and set the overflow flag defined by the binding) instead of raising Constraint_Error in overflow situations, when traps are disabled. Similarly, it is appropriate for the complex elementary functions to deliver results with an infinite component (and set the zero-divide flag defined by the binding) instead of raising Constraint_Error at poles, when traps are disabled. Finally, such a binding should also specify the behavior of the complex elementary functions, when sensible, given parameters with infinite components.

Implementation Requirements

34 In the implementation of Numerics.Generic_Complex_Elementary_Functions, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Complex_Types.Real.

34.a **Implementation Note:** Implementations of Numerics.Generic_Complex_Elementary_Functions written in Ada should therefore avoid declaring local variables of subtype Complex_Types.Real; the subtype Complex_Types.Real'Base should be used instead.

35 {*prescribed result (for the evaluation of a complex elementary function)*} In the following cases, evaluation of a complex elementary function shall yield the *prescribed result* (or a result having the prescribed component), provided that the preceding rules do not call for an exception to be raised:

36 • When the parameter X has the value zero, the Sqrt, Sin, Arcsin, Tan, Arctan, Sinh, Arcsinh, Tanh, and Arctanh functions yield a result of zero; the Exp, Cos, and Cosh functions yield a result of one; the Arccos and Arccot functions yield a real result; and the Arccoth function yields an imaginary result.

37 • When the parameter X has the value one, the Sqrt function yields a result of one; the Log, Arccos, and Arccosh functions yield a result of zero; and the Arcsin function yields a real result.

38 • When the parameter X has the value –1.0, the Sqrt function yields the result

39   • *i* (resp., –*i*), when the sign of the imaginary component of X is positive (resp., negative), if Complex_Types.Real'Signed_Zeros is True;

40   • *i*, if Complex_Types.Real'Signed_Zeros is False;

41/2 • {*AI95-00434-01*} When the parameter X has the value –1.0, the Log function yields an imaginary result; and the Arcsin and Arccos functions yield a real result.

42 • When the parameter X has the value ± *i*, the Log function yields an imaginary result.

43 • Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand (as a complex value). Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero.

43.a **Discussion:** It is possible to give many other prescribed results restricting the result to the real or imaginary axis when the parameter X is appropriately restricted to easily testable portions of the domain. We follow the proposed ISO/IEC standard for Generic_Complex_Elementary_Functions (for Ada 83), CD 13813, in not doing so, however.

Other accuracy requirements for the complex elementary functions, which apply only in the strict mode, are given in G.2.6. 44

The sign of a zero result or zero result component yielded by a complex elementary function is implementation defined when Complex_Types.Real'Signed_Zeros is True. 45

Implementation defined: The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Elementary_Functions, when Complex_Types.Real'Signed_Zeros is True. 45.a

Implementation Permissions

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package with the appropriate predefined nongeneric equivalent of Numerics.Generic_Complex_Types; if they are, then the latter shall have been obtained by actual instantiation of Numerics.Generic_Complex_Types. 46

The exponentiation operator may be implemented in terms of the Exp and Log functions. Because this implementation yields poor accuracy in some parts of the domain, no accuracy requirement is imposed on complex exponentiation. 47

{*unspecified* [partial]} The implementation of the Exp function of a complex parameter X is allowed to raise the exception Constraint_Error, signaling overflow, when the real component of X exceeds an unspecified threshold that is approximately log(Complex_Types.Real'Safe_Last). This permission recognizes the impracticality of avoiding overflow in the marginal case that the exponential of the real component of X exceeds the safe range of Complex_Types.Real but both components of the final result do not. Similarly, the Sin and Cos (resp., Sinh and Cosh) functions are allowed to raise the exception Constraint_Error, signaling overflow, when the absolute value of the imaginary (resp., real) component of the parameter X exceeds an unspecified threshold that is approximately log(Complex_Types.Real'Safe_Last) + log(2.0). {*unspecified* [partial]} This permission recognizes the impracticality of avoiding overflow in the marginal case that the hyperbolic sine or cosine of the imaginary (resp., real) component of X exceeds the safe range of Complex_Types.Real but both components of the final result do not. 48

Implementation Advice

Implementations in which Complex_Types.Real'Signed_Zeros is True should attempt to provide a rational treatment of the signs of zero results and result components. For example, many of the complex elementary functions have components that are odd functions of one of the parameter components; in these cases, the result component should have the sign of the parameter component at the origin. Other complex elementary functions have zero components whose sign is opposite that of a parameter component at the origin, or is always positive or always negative. 49

Implementation Advice: If Complex_Types.Real'Signed_Zeros is true for Numerics.Generic_Complex_Elementary_-Functions, a rational treatment of the signs of zero results and result components should be provided. 49.a.1/2

Wording Changes from Ada 83

The semantics of Numerics.Generic_Complex_Elementary_Functions differs from Generic_Complex_Elementary_-Functions as defined in ISO/IEC CD 13814 (for Ada 83) in the following ways: 49.a

- The generic package is a child unit of the package defining the Argument_Error exception. 49.b
- The proposed Generic_Complex_Elementary_Functions standard (for Ada 83) specified names for the nongeneric equivalents, if provided. Here, those nongeneric equivalents are required. 49.c
- The generic package imports an instance of Numerics.Generic_Complex_Types rather than a long list of individual types and operations exported by such an instance. 49.d
- The dependence of the imaginary component of the Sqrt and Log functions on the sign of a zero parameter component is tied to the value of Complex_Types.Real'Signed_Zeros. 49.e
- Conformance to accuracy requirements is conditional. 49.f

Wording Changes from Ada 95

49.g/2 {*8652/0020*} {*AI95-00126-01*} **Corrigendum:** Explicitly stated that the nongeneric equivalents of Generic_Complex_Elementary_Functions are pure.

49.h/2 {*AI95-00185-01*} Corrected various inconsistencies in the definition of the branch cuts.

G.1.3 Complex Input-Output

1 The generic package Text_IO.Complex_IO defines procedures for the formatted input and output of complex values. The generic actual parameter in an instantiation of Text_IO.Complex_IO is an instance of Numerics.Generic_Complex_Types for some floating point subtype. Exceptional conditions are reported by raising the appropriate exception defined in Text_IO.

1.a **Implementation Note:** An implementation of Text_IO.Complex_IO can be built around an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, where Complex_Types is the generic formal package parameter of Text_IO.Complex_IO. There is no need for an implementation of Text_IO.Complex_IO to parse real values.

Static Semantics

2 The generic library package Text_IO.Complex_IO has the following declaration:

2.a **Ramification:** Because this is a child of Text_IO, the declarations of the visible part of Text_IO are directly visible within it.

3
```
with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (<>);
package Ada.Text_IO.Complex_IO is
```
4
```
   use Complex_Types;
```
5
```
   Default_Fore : Field := 2;
   Default_Aft  : Field := Real'Digits - 1;
   Default_Exp  : Field := 3;
```
6
```
   procedure Get (File  : in  File_Type;
                  Item  : out Complex;
                  Width : in  Field := 0);
   procedure Get (Item  : out Complex;
                  Width : in  Field := 0);
```
7
```
   procedure Put (File : in File_Type;
                  Item : in Complex;
                  Fore : in Field := Default_Fore;
                  Aft  : in Field := Default_Aft;
                  Exp  : in Field := Default_Exp);
   procedure Put (Item : in Complex;
                  Fore : in Field := Default_Fore;
                  Aft  : in Field := Default_Aft;
                  Exp  : in Field := Default_Exp);
```
8
```
   procedure Get (From : in  String;
                  Item : out Complex;
                  Last : out Positive);
   procedure Put (To   : out String;
                  Item : in  Complex;
                  Aft  : in  Field := Default_Aft;
                  Exp  : in  Field := Default_Exp);
```
9
```
end Ada.Text_IO.Complex_IO;
```

9.1/2 {*AI95-00328-01*} The library package Complex_Text_IO defines the same subprograms as Text_IO.Complex_IO, except that the predefined type Float is systematically substituted for Real, and the type Numerics.Complex_Types.Complex is systematically substituted for Complex throughout. Nongeneric equivalents of Text_IO.Complex_IO corresponding to each of the other predefined floating point types are defined similarly, with the names Short_Complex_Text_IO, Long_Complex_Text_IO, etc.

Reason: The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics. 9.a/2

The semantics of the Get and Put procedures are as follows: 10

11
```
procedure Get (File  : in  File_Type;
               Item  : out Complex;
               Width : in  Field := 0);
procedure Get (Item  : out Complex;
               Width : in  Field := 0);
```

12/1 {*8652/0092*} {*AI95-00029-01*} The input sequence is a pair of optionally signed real literals representing the real and imaginary components of a complex value These components have the format defined for the corresponding Get procedure of an instance of Text_IO.Float_IO (see A.10.9) for the base subtype of Complex_Types.Real. The pair of components may be separated by a comma or surrounded by a pair of parentheses or both. Blanks are freely allowed before each of the components and before the parentheses and comma, if either is used. If the value of the parameter Width is zero, then

- line and page terminators are also allowed in these places; 13
- the components shall be separated by at least one blank or line terminator if the comma is omitted; and 14
- reading stops when the right parenthesis has been read, if the input sequence includes a left parenthesis, or when the imaginary component has been read, otherwise. 15

If a nonzero value of Width is supplied, then 15.1

- the components shall be separated by at least one blank if the comma is omitted; and 16
- exactly Width characters are read, or the characters (possibly none) up to a line terminator, whichever comes first (blanks are included in the count). 17

Reason: The parenthesized and comma-separated form is the form produced by Put on output (see below), and also by list-directed output in Fortran. The other allowed forms match several common styles of edit-directed output in Fortran, allowing most preexisting Fortran data files containing complex data to be read easily. When such files contain complex values with no separation between the real and imaginary components, the user will have to read those components separately, using an instance of Text_IO.Float_IO. 17.a

Returns, in the parameter Item, the value of type Complex that corresponds to the input sequence. 18

The exception Text_IO.Data_Error is raised if the input sequence does not have the required syntax or if the components of the complex value obtained are not of the base subtype of Complex_Types.Real. 19

20
```
procedure Put (File : in File_Type;
               Item : in Complex;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
procedure Put (Item : in Complex;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
```

Outputs the value of the parameter Item as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically, 21

- outputs a left parenthesis; 22

23 • outputs the value of the real component of the parameter Item with the format defined by the corresponding Put procedure of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using the given values of Fore, Aft, and Exp;

24 • outputs a comma;

25 • outputs the value of the imaginary component of the parameter Item with the format defined by the corresponding Put procedure of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using the given values of Fore, Aft, and Exp;

26 • outputs a right parenthesis.

26.a **Discussion:** If the file has a bounded line length, a line terminator may be output implicitly before any element of the sequence itemized above.

26.b **Discussion:** The option of outputting the complex value as a pair of reals without additional punctuation is not provided, since it can be accomplished by outputting the real and imaginary components of the complex value separately.

27
```
procedure Get (From : in  String;
               Item : out Complex;
               Last : out Positive);
```

28/2 {*AI95-00434-01*} Reads a complex value from the beginning of the given string, following the same rule as the Get procedure that reads a complex value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Complex that corresponds to the input sequence. Returns in Last the index value such that From(Last) is the last character read.

29 The exception Text_IO.Data_Error is raised if the input sequence does not have the required syntax or if the components of the complex value obtained are not of the base subtype of Complex_Types.Real.

30
```
procedure Put (To   : out String;
               Item : in  Complex;
               Aft  : in  Field := Default_Aft;
               Exp  : in  Field := Default_Exp);
```

31 Outputs the value of the parameter Item to the given string as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically,

32 • a left parenthesis, the real component, and a comma are left justified in the given string, with the real component having the format defined by the Put procedure (for output to a file) of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using a value of zero for Fore and the given values of Aft and Exp;

33 • the imaginary component and a right parenthesis are right justified in the given string, with the imaginary component having the format defined by the Put procedure (for output to a file) of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using a value for Fore that completely fills the remainder of the string, together with the given values of Aft and Exp.

33.a **Reason:** This rule is the one proposed in LSN-1051. Other rules were considered, including one that would have read "Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using a value for Fore such that the sequence of characters output exactly fills, or comes closest to filling, the string; in the latter case, the string is filled by inserting one extra blank immediately after the comma." While this latter rule might be considered the closest analogue to the rule for output to a string in Text_IO.Float_IO, it requires a more difficult and inefficient implementation involving special cases when the integer part of one component is substantially longer than that of the other and the string is too short to allow both to be preceded by blanks. Unless such a special case applies, the latter rule might produce better columnar output if several such strings are ultimately output to a file, but very nearly the same output can be produced by outputting to the file directly, with the appropriate value of Fore; in any case, it might validly be assumed that output to a string is intended for further computation rather than for display, so that the precise formatting of the string to achieve a particular appearance is not the major concern.

The exception Text_IO.Layout_Error is raised if the given string is too short to hold the formatted output. 34

Implementation Permissions

Other exceptions declared (by renaming) in Text_IO may be raised by the preceding procedures in the appropriate circumstances, as for the corresponding procedures of Text_IO.Float_IO. 35

Extensions to Ada 95

{*AI95-00328-01*} {*extensions to Ada 95*} Nongeneric equivalents for Text_IO.Complex_IO are added, to be consistent with all other language-defined Numerics generic packages. 35.a/2

Wording Changes from Ada 95

{*8652/0092*} {*AI95-00029-01*} **Corrigendum:** Clarified that the syntax of values read by Complex_IO is the same as that read by Text_IO.Float_IO. 35.b/2

G.1.4 The Package Wide_Text_IO.Complex_IO

Static Semantics

{*Ada.Wide_*} Implementations shall also provide the generic library package Wide_Text_IO.Complex_IO. Its declaration is obtained from that of Text_IO.Complex_IO by systematically replacing Text_IO by Wide_Text_IO and String by Wide_String; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide characters. 1

G.1.5 The Package Wide_Wide_Text_IO.Complex_IO

Static Semantics

{*AI95-00285-01*} {*Ada.Wide_Wide_*} Implementations shall also provide the generic library package Wide_Wide_Text_IO.Complex_IO. Its declaration is obtained from that of Text_IO.Complex_IO by systematically replacing Text_IO by Wide_Wide_Text_IO and String by Wide_Wide_String; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide wide characters. 1/2

Extensions to Ada 95

{*AI95-00285-01*} {*extensions to Ada 95*} Package Wide_Wide_Text_IO.Complex_IO is new. (At least it wasn't called Incredibly_Wide_Text_IO.Complex_IO; maybe next time.) 1.a/2

G.2 Numeric Performance Requirements

Implementation Requirements

{*accuracy*} {*strict mode*} Implementations shall provide a user-selectable mode in which the accuracy and other numeric performance requirements detailed in the following subclauses are observed. This mode, referred to as the *strict mode*, may or may not be the default mode; it directly affects the results of the predefined arithmetic operations of real types and the results of the subprograms in children of the Numerics package, and indirectly affects the operations in other language defined packages. {*relaxed mode*} Implementations shall also provide the opposing mode, which is known as the *relaxed mode*. 1

1.a **Reason:** On the assumption that the users of an implementation that does not support the Numerics Annex have no particular need for numerical performance, such an implementation has no obligation to meet any particular requirements in this area. On the other hand, users of an implementation that does support the Numerics Annex are provided with a way of ensuring that their programs achieve a known level of numerical performance and that the performance is portable to other such implementations. The relaxed mode is provided to allow implementers to offer an efficient but not fully accurate alternative in the case that the strict mode entails a time overhead that some users may find excessive. In some of its areas of impact, the relaxed mode may be fully equivalent to the strict mode.

1.b **Implementation Note:** The relaxed mode may, for example, be used to exploit the implementation of (some of) the elementary functions in hardware, when available. Such implementations often do not meet the accuracy requirements of the strict mode, or do not meet them over the specified range of parameter values, but compensate in other ways that may be important to the user, such as their extreme speed.

1.c **Ramification:** For implementations supporting the Numerics Annex, the choice of mode has no effect on the selection of a representation for a real type or on the values of attributes of a real type.

Implementation Permissions

2 Either mode may be the default mode.

2.a **Implementation defined:** Whether the strict mode or the relaxed mode is the default.

3 The two modes need not actually be different.

Extensions to Ada 83

3.a {*extensions to Ada 83*} The choice between strict and relaxed numeric performance was not available in Ada 83.

G.2.1 Model of Floating Point Arithmetic

1 In the strict mode, the predefined operations of a floating point type shall satisfy the accuracy requirements specified here and shall avoid or signal overflow in the situations described. This behavior is presented in terms of a model of floating point arithmetic that builds on the concept of the canonical form (see A.5.3).

Static Semantics

2 Associated with each floating point type is an infinite set of model numbers. The model numbers of a type are used to define the accuracy requirements that have to be satisfied by certain predefined operations of the type; through certain attributes of the model numbers, they are also used to explain the meaning of a user-declared floating point type declaration. The model numbers of a derived type are those of the parent type; the model numbers of a subtype are those of its type.

3 {*model number*} The *model numbers* of a floating point type T are zero and all the values expressible in the canonical form (for the type T), in which *mantissa* has T'Model_Mantissa digits and *exponent* has a value greater than or equal to T'Model_Emin. (These attributes are defined in G.2.2.)

3.a **Discussion:** The model is capable of describing the behavior of most existing hardware that has a mantissa-exponent representation. As applied to a type T, it is parameterized by the values of T'Machine_Radix, T'Model_Mantissa, T'Model_Emin, T'Safe_First, and T'Safe_Last. The values of these attributes are determined by how, and how well, the hardware behaves. They in turn determine the set of model numbers and the safe range of the type, which figure in the accuracy and range (overflow avoidance) requirements.

3.b In hardware that is free of arithmetic anomalies, T'Model_Mantissa, T'Model_Emin, T'Safe_First, and T'Safe_Last will yield the same values as T'Machine_Mantissa, T'Machine_Emin, T'Base'First, and T'Base'Last, respectively, and the model numbers in the safe range of the type T will coincide with the machine numbers of the type T. In less perfect hardware, it is not possible for the model-oriented attributes to have these optimal values, since the hardware, by definition, and therefore the implementation, cannot conform to the stringencies of the resulting model; in this case, the values yielded by the model-oriented parameters have to be made more conservative (i.e., have to be penalized), with the result that the model numbers are more widely separated than the machine numbers, and the safe range is a subrange of the base range. The implementation will then be able to conform to the requirements of the weaker model defined by the sparser set of model numbers and the smaller safe range.

{*model interval*} A *model interval* of a floating point type is any interval whose bounds are model numbers of the type. {*model interval (associated with a value)*} The *model interval* of a type T *associated with a value v* is the smallest model interval of T that includes *v*. (The model interval associated with a model number of a type consists of that number only.) 4

Implementation Requirements

The accuracy requirements for the evaluation of certain predefined operations of floating point types are as follows. 5

Discussion: This subclause does not cover the accuracy of an operation of a static expression; such operations have to be evaluated exactly (see 4.9). It also does not cover the accuracy of the predefined attributes of a floating point subtype that yield a value of the type; such operations also yield exact results (see 3.5.8 and A.5.3). 5.a

{*operand interval*} An *operand interval* is the model interval, of the type specified for the operand of an operation, associated with the value of the operand. 6

For any predefined arithmetic operation that yields a result of a floating point type T, the required bounds on the result are given by a model interval of T (called the *result interval*) defined in terms of the operand values as follows: 7

- {*result interval (for the evaluation of a predefined arithmetic operation)*} The result interval is the smallest model interval of T that includes the minimum and the maximum of all the values obtained by applying the (exact) mathematical operation to values arbitrarily selected from the respective operand intervals. 8

The result interval of an exponentiation is obtained by applying the above rule to the sequence of multiplications defined by the exponent, assuming arbitrary association of the factors, and to the final division in the case of a negative exponent. 9

The result interval of a conversion of a numeric value to a floating point type T is the model interval of T associated with the operand value, except when the source expression is of a fixed point type with a *small* that is not a power of T'Machine_Radix or is a fixed point multiplication or division either of whose operands has a *small* that is not a power of T'Machine_Radix; in these cases, the result interval is implementation defined. 10

Implementation defined: The result interval in certain cases of fixed-to-float conversion. 10.a

{*Overflow_Check* [partial]} {*check, language-defined (Overflow_Check)*} For any of the foregoing operations, the implementation shall deliver a value that belongs to the result interval when both bounds of the result interval are in the safe range of the result type T, as determined by the values of T'Safe_First and T'Safe_Last; otherwise, 11

- {*Constraint_Error (raised by failure of run-time check)*} if T'Machine_Overflows is True, the implementation shall either deliver a value that belongs to the result interval or raise Constraint_Error; 12
- if T'Machine_Overflows is False, the result is implementation defined. 13

Implementation defined: The result of a floating point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. 13.a

For any predefined relation on operands of a floating point type T, the implementation may deliver any value (i.e., either True or False) obtained by applying the (exact) mathematical comparison to values arbitrarily chosen from the respective operand intervals. 14

The result of a membership test is defined in terms of comparisons of the operand value with the lower and upper bounds of the given range or type mark (the usual rules apply to these comparisons). 15

Implementation Permissions

16 If the underlying floating point hardware implements division as multiplication by a reciprocal, the result interval for division (and exponentiation by a negative exponent) is implementation defined.

16.a **Implementation defined:** The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal.

Wording Changes from Ada 83

16.b The Ada 95 model numbers of a floating point type that are in the safe range of the type are comparable to the Ada 83 safe numbers of the type. There is no analog of the Ada 83 model numbers. The Ada 95 model numbers, when not restricted to the safe range, are an infinite set.

Inconsistencies With Ada 83

16.c {*inconsistencies with Ada 83*} Giving the model numbers the hardware radix, instead of always a radix of two, allows (in conjunction with other changes) some borderline declared types to be represented with less precision than in Ada 83 (i.e., with single precision, whereas Ada 83 would have used double precision). Because the lower precision satisfies the requirements of the model (and did so in Ada 83 as well), this change is viewed as a desirable correction of an anomaly, rather than a worrisome inconsistency. (Of course, the wider representation chosen in Ada 83 also remains eligible for selection in Ada 95.)

16.d As an example of this phenomenon, assume that Float is represented in single precision and that a double precision type is also available. Also assume hexadecimal hardware with clean properties, for example certain IBM hardware. Then,

16.e
```
type T is digits Float'Digits range -Float'Last .. Float'Last;
```

16.f results in T being represented in double precision in Ada 83 and in single precision in Ada 95. The latter is intuitively correct; the former is counterintuitive. The reason why the double precision type is used in Ada 83 is that Float has model and safe numbers (in Ada 83) with 21 binary digits in their mantissas, as is required to model the hypothesized hexadecimal hardware using a binary radix; thus Float'Last, which is not a model number, is slightly outside the range of safe numbers of the single precision type, making that type ineligible for selection as the representation of T even though it provides adequate precision. In Ada 95, Float'Last (the same value as before) is a model number and is in the safe range of Float on the hypothesized hardware, making Float eligible for the representation of T.

Extensions to Ada 83

16.g {*extensions to Ada 83*} Giving the model numbers the hardware radix allows for practical implementations on decimal hardware.

Wording Changes from Ada 83

16.h The wording of the model of floating point arithmetic has been simplified to a large extent.

G.2.2 Model-Oriented Attributes of Floating Point Types

1 In implementations that support the Numerics Annex, the model-oriented attributes of floating point types shall yield the values defined here, in both the strict and the relaxed modes. These definitions add conditions to those in A.5.3.

Static Semantics

2 For every subtype S of a floating point type *T*:

3/2 {*AI95-00256-01*} S'Model_Mantissa

Yields the number of digits in the mantissa of the canonical form of the model numbers of *T* (see A.5.3). The value of this attribute shall be greater than or equal to

3.1/2 $\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix}) \rceil + g$

3.2/2 where *d* is the requested decimal precision of *T*, and *g* is 0 if *T*'Machine_Radix is a positive power of 10 and 1 otherwise. In addition, *T*'Model_Mantissa shall be less than or equal to the value of *T*'Machine_Mantissa. This attribute yields a value of the type *universal_integer*.

Ramification: S'Model_Epsilon, which is defined in terms of S'Model_Mantissa (see A.5.3), yields the absolute value of the difference between one and the next model number of the type *T* above one. It is equal to or larger than the absolute value of the difference between one and the next machine number of the type *T* above one. 3.a

S'Model_Emin 4

Yields the minimum exponent of the canonical form of the model numbers of *T* (see A.5.3). The value of this attribute shall be greater than or equal to the value of *T*'Machine_Emin. This attribute yields a value of the type *universal_integer*.

Ramification: S'Model_Small, which is defined in terms of S'Model_Emin (see A.5.3), yields the smallest positive (nonzero) model number of the type *T*. 4.a

S'Safe_First 5

Yields the lower bound of the safe range of *T*. The value of this attribute shall be a model number of *T* and greater than or equal to the lower bound of the base range of *T*. In addition, if *T* is declared by a floating_point_definition or is derived from such a type, and the floating_point_definition includes a real_range_specification specifying a lower bound of *lb*, then the value of this attribute shall be less than or equal to *lb*; otherwise, it shall be less than or equal to $-10.0^{4 \cdot d}$, where *d* is the requested decimal precision of *T*. This attribute yields a value of the type *universal_real*.

S'Safe_Last 6

Yields the upper bound of the safe range of *T*. The value of this attribute shall be a model number of *T* and less than or equal to the upper bound of the base range of *T*. In addition, if *T* is declared by a floating_point_definition or is derived from such a type, and the floating_point_definition includes a real_range_specification specifying an upper bound of *ub*, then the value of this attribute shall be greater than or equal to *ub*; otherwise, it shall be greater than or equal to $10.0^{4 \cdot d}$, where d is the requested decimal precision of *T*. This attribute yields a value of the type *universal_real*.

{*Constraint_Error (raised by failure of run-time check)*} S'Model Denotes a function (of a parameter *X*) whose specification is given in A.5.3. If *X* is a model number of *T*, the function yields *X*; otherwise, it yields the value obtained by rounding or truncating *X* to either one of the adjacent model numbers of *T*. {*Overflow_Check* [partial]} {*check, language-defined (Overflow_Check)*} Constraint_Error is raised if the resulting model number is outside the safe range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. 7

Subject to the constraints given above, the values of S'Model_Mantissa and S'Safe_Last are to be maximized, and the values of S'Model_Emin and S'Safe_First minimized, by the implementation as follows: 8

- First, S'Model_Mantissa is set to the largest value for which values of S'Model_Emin, S'Safe_First, and S'Safe_Last can be chosen so that the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes. 9
- Next, S'Model_Emin is set to the smallest value for which values of S'Safe_First and S'Safe_Last can be chosen so that the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes and the previously determined value of S'Model_Mantissa. 10
- Finally, S'Safe_First and S'Safe_last are set (in either order) to the smallest and largest values, respectively, for which the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes and the previously determined values of S'Model_Mantissa and S'Model_Emin. 11

Ramification: {*IEEE floating point arithmetic*} {*IEC 559:1989*} The following table shows appropriate attribute values for IEEE basic single and double precision types (ANSI/IEEE Std 754-1985, IEC 559:1989). Here, we use the names IEEE_Float_32 and IEEE_Float_64, the names that would typically be declared in package Interfaces, in an implementation that supports IEEE arithmetic. In such an implementation, the attributes would typically be the same for Standard.Float and Long_Float, respectively. 11.a

```
11.b      Attribute                          IEEE_Float_32                IEEE_Float_64
11.c      'Machine_Radix                                 2                            2
          'Machine_Mantissa                             24                           53
          'Machine_Emin                               -125                        -1021
          'Machine_Emax                                128                         1024
          'Denorm                                     True                         True
          'Machine_Rounds                             True                         True
          'Machine_Overflows                    True/False                   True/False
          'Signed_Zeros                     should be True               should be True
11.d      'Model_Mantissa    (same as 'Machine_Mantissa)  (same as 'Machine_Mantissa)
          'Model_Emin            (same as 'Machine_Emin)      (same as 'Machine_Emin)
          'Model_Epsilon                      2.0**(-23)                   2.0**(-52)
          'Model_Small                       2.0**(-126)                 2.0**(-1022)
          'Safe_First        -2.0**128*(1.0-2.0**(-24))  -2.0**1024*(1.0-2.0**(-53))
          'Safe_Last          2.0**128*(1.0-2.0**(-24))   2.0**1024*(1.0-2.0**(-53))
11.e      'Digits                                    6                           15
          'Base'Digits                (same as 'Digits)            (same as 'Digits)
11.f      'First                  (same as 'Safe_First)        (same as 'Safe_First)
          'Last                    (same as 'Safe_Last)         (same as 'Safe_Last)
          'Size                                     32                           64
```

11.g Note: 'Machine_Overflows can be True or False, depending on whether the Ada implementation raises Constraint_Error or delivers a signed infinity in overflow and zerodivide situations (and at poles of the elementary functions).

Wording Changes from Ada 95

11.h/2 {*AI95-00256-01*} Corrected the definition of Model_Mantissa to match that given in 3.5.8.

G.2.3 Model of Fixed Point Arithmetic

1 In the strict mode, the predefined arithmetic operations of a fixed point type shall satisfy the accuracy requirements specified here and shall avoid or signal overflow in the situations described.

Implementation Requirements

2 The accuracy requirements for the predefined fixed point arithmetic operations and conversions, and the results of relations on fixed point operands, are given below.

2.a **Discussion:** This subclause does not cover the accuracy of an operation of a static expression; such operations have to be evaluated exactly (see 4.9).

3 The operands of the fixed point adding operators, absolute value, and comparisons have the same type. These operations are required to yield exact results, unless they overflow.

4 Multiplications and divisions are allowed between operands of any two fixed point types; the result has to be (implicitly or explicitly) converted to some other numeric type. For purposes of defining the accuracy rules, the multiplication or division and the conversion are treated as a single operation whose accuracy depends on three types (those of the operands and the result). For decimal fixed point types, the attribute T'Round may be used to imply explicit conversion with rounding (see 3.5.10).

5 When the result type is a floating point type, the accuracy is as given in G.2.1. {*perfect result set*} For some combinations of the operand and result types in the remaining cases, the result is required to belong to a small set of values called the *perfect result set*; {*close result set*} for other combinations, it is required merely to belong to a generally larger and implementation-defined set of values called the *close result set*. When the result type is a decimal fixed point type, the perfect result set contains a single value; thus, operations on decimal types are always fully specified.

5.a **Implementation defined:** The definition of *close result set*, which determines the accuracy of certain fixed point multiplications and divisions.

6 When one operand of a fixed-fixed multiplication or division is of type *universal_real*, that operand is not implicitly converted in the usual sense, since the context does not determine a unique target type, but the

accuracy of the result of the multiplication or division (i.e., whether the result has to belong to the perfect result set or merely the close result set) depends on the value of the operand of type *universal_real* and on the types of the other operand and of the result.

6.a **Discussion:** We need not consider here the multiplication or division of two such operands, since in that case either the operation is evaluated exactly (i.e., it is an operation of a static expression all of whose operators are of a root numeric type) or it is considered to be an operation of a floating point type.

7 For a fixed point multiplication or division whose (exact) mathematical result is *v*, and for the conversion of a value *v* to a fixed point type, the perfect result set and close result set are defined as follows:

- 8 If the result type is an ordinary fixed point type with a *small* of *s*,
  - 9 if *v* is an integer multiple of *s*, then the perfect result set contains only the value *v*;
  - 10 otherwise, it contains the integer multiple of *s* just below *v* and the integer multiple of *s* just above *v*.

  11 The close result set is an implementation-defined set of consecutive integer multiples of *s* containing the perfect result set as a subset.

- 12 If the result type is a decimal type with a *small* of *s*,
  - 13 if *v* is an integer multiple of *s*, then the perfect result set contains only the value *v*;
  - 14 otherwise, if truncation applies then it contains only the integer multiple of *s* in the direction toward zero, whereas if rounding applies then it contains only the nearest integer multiple of *s* (with ties broken by rounding away from zero).

  15 The close result set is an implementation-defined set of consecutive integer multiples of *s* containing the perfect result set as a subset.

  15.a **Ramification:** As a consequence of subsequent rules, this case does not arise when the operand types are also decimal types.

- 16 If the result type is an integer type,
  - 17 if *v* is an integer, then the perfect result set contains only the value *v*;
  - 18 otherwise, it contains the integer nearest to the value *v* (if *v* lies equally distant from two consecutive integers, the perfect result set contains the one that is further from zero).

  19 The close result set is an implementation-defined set of consecutive integers containing the perfect result set as a subset.

20 The result of a fixed point multiplication or division shall belong either to the perfect result set or to the close result set, as described below, if overflow does not occur. In the following cases, if the result type is a fixed point type, let *s* be its *small*; otherwise, i.e. when the result type is an integer type, let *s* be 1.0.

- 21 For a multiplication or division neither of whose operands is of type *universal_real*, let *l* and *r* be the *smalls* of the left and right operands. For a multiplication, if $(l \cdot r) / s$ is an integer or the reciprocal of an integer (the *smalls* are said to be “compatible” in this case), the result shall belong to the perfect result set; otherwise, it belongs to the close result set. For a division, if $l / (r \cdot s)$ is an integer or the reciprocal of an integer (i.e., the *smalls* are compatible), the result shall belong to the perfect result set; otherwise, it belongs to the close result set.

  21.a **Ramification:** When the operand and result types are all decimal types, their *smalls* are necessarily compatible; the same is true when they are all ordinary fixed point types with binary *smalls*.

- 22 For a multiplication or division having one *universal_real* operand with a value of *v*, note that it is always possible to factor *v* as an integer multiple of a “compatible” *small*, but the integer multiple may be “too big.” If there exists a factorization in which that multiple is less than some implementation-defined limit, the result shall belong to the perfect result set; otherwise, it belongs to the close result set.

22.a **Implementation defined:** Conditions on a *universal_real* operand of a fixed point multiplication or division for which the result shall be in the *perfect result set*.

23 A multiplication P * Q of an operand of a fixed point type F by an operand of an integer type I, or vice-versa, and a division P / Q of an operand of a fixed point type F by an operand of an integer type I, are also allowed. In these cases, the result has a type of F; explicit conversion of the result is never required. The accuracy required in these cases is the same as that required for a multiplication F(P * Q) or a division F(P / Q) obtained by interpreting the operand of the integer type to have a fixed point type with a *small* of 1.0.

24 The accuracy of the result of a conversion from an integer or fixed point type to a fixed point type, or from a fixed point type to an integer type, is the same as that of a fixed point multiplication of the source value by a fixed point operand having a *small* of 1.0 and a value of 1.0, as given by the foregoing rules. The result of a conversion from a floating point type to a fixed point type shall belong to the close result set. The result of a conversion of a *universal_real* operand to a fixed point type shall belong to the perfect result set.

25 The possibility of overflow in the result of a predefined arithmetic operation or conversion yielding a result of a fixed point type T is analogous to that for floating point types, except for being related to the base range instead of the safe range. {*Overflow_Check* [partial]} {*check, language-defined (Overflow_Check)*} If all of the permitted results belong to the base range of T, then the implementation shall deliver one of the permitted results; otherwise,

26
- {*Constraint_Error (raised by failure of run-time check)*} if T'Machine_Overflows is True, the implementation shall either deliver one of the permitted results or raise Constraint_Error;

27
- if T'Machine_Overflows is False, the result is implementation defined.

27.a **Implementation defined:** The result of a fixed point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False.

Inconsistencies With Ada 83

27.b {*inconsistencies with Ada 83*} Since the values of a fixed point type are now just the integer multiples of its *small*, the possibility of using extra bits available in the chosen representation for extra accuracy rather than for increasing the base range would appear to be removed, raising the possibility that some fixed point expressions will yield less accurate results than in Ada 83. However, this is partially offset by the ability of an implementation to choose a smaller default *small* than before. Of course, if it does so for a type T then T'Small will have a different value than it previously had.

27.c The accuracy requirements in the case of incompatible *smalls* are relaxed to foster wider support for non-binary *smalls*. If this relaxation is exploited for a type that was previously supported, lower accuracy could result; however, there is no particular incentive to exploit the relaxation in such a case.

Wording Changes from Ada 83

27.d The fixed point accuracy requirements are now expressed without reference to model or safe numbers, largely because the full generality of the former model was never exploited in the case of fixed point types (particularly in regard to operand perturbation). Although the new formulation in terms of perfect result sets and close result sets is still verbose, it can be seen to distill down to two cases:

27.e
- a case where the result must be the exact result, if the exact result is representable, or, if not, then either one of the adjacent values of the type (in some subcases only one of those adjacent values is allowed);

27.f
- a case where the accuracy is not specified by the language.

G.2.4 Accuracy Requirements for the Elementary Functions

1 In the strict mode, the performance of Numerics.Generic_Elementary_Functions shall be as specified here.

Implementation Requirements

2 {*result interval (for the evaluation of an elementary function)*} {*maximum relative error (for the evaluation of an elementary function)*} When an exception is not raised, the result of evaluating a function in an instance *EF* of Numerics.Generic_Elementary_Functions belongs to a *result interval*, defined as the smallest model interval of *EF*.Float_Type that contains all the values of the form $f \cdot (1.0 + d)$, where *f* is the exact value of the corresponding mathematical function at the given parameter values, *d* is a real number, and $|d|$ is less than or equal to the function's *maximum relative error*. {*Overflow_Check* [partial]} {*check, language-defined (Overflow_Check)*} The function delivers a value that belongs to the result interval when both of its bounds belong to the safe range of *EF*.Float_Type; otherwise,

- 3 {*Constraint_Error (raised by failure of run-time check)*} if *EF*.Float_Type'Machine_Overflows is True, the function either delivers a value that belongs to the result interval or raises Constraint_Error, signaling overflow;
- 4 if *EF*.Float_Type'Machine_Overflows is False, the result is implementation defined.

4.a **Implementation defined:** The result of an elementary function reference in overflow situations, when the Machine_Overflows attribute of the result type is False.

5 The maximum relative error exhibited by each function is as follows:

- 6 2.0 · *EF*.Float_Type'Model_Epsilon, in the case of the Sqrt, Sin, and Cos functions;
- 7 4.0 · *EF*.Float_Type'Model_Epsilon, in the case of the Log, Exp, Tan, Cot, and inverse trigonometric functions; and
- 8 8.0 · *EF*.Float_Type'Model_Epsilon, in the case of the forward and inverse hyperbolic functions.

9 The maximum relative error exhibited by the exponentiation operator, which depends on the values of the operands, is (4.0 + |Right · log(Left)| / 32.0) · *EF*.Float_Type'Model_Epsilon.

10 The maximum relative error given above applies throughout the domain of the forward trigonometric functions when the Cycle parameter is specified. {*angle threshold*} When the Cycle parameter is omitted, the maximum relative error given above applies only when the absolute value of the angle parameter X is less than or equal to some implementation-defined *angle threshold*, which shall be at least *EF*.Float_Type'Machine_Radix $^{\lfloor EF\text{.Float\_Type'Machine\_Mantissa}/2\rfloor}$. Beyond the angle threshold, the accuracy of the forward trigonometric functions is implementation defined.

10.a **Implementation defined:** The value of the *angle threshold*, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound.

10.b **Implementation defined:** The accuracy of certain elementary functions for parameters beyond the angle threshold.

10.c **Implementation Note:** The angle threshold indirectly determines the amount of precision that the implementation has to maintain during argument reduction.

11/2 {*AI95-00434-01*} The prescribed results specified in A.5.1 for certain functions at particular parameter values take precedence over the maximum relative error bounds; effectively, they narrow to a single value the result interval allowed by the maximum relative error bounds. Additional rules with a similar effect are given by table G-1 for the inverse trigonometric functions, at particular parameter values for which the mathematical result is possibly not a model number of *EF*.Float_Type (or is, indeed, even transcendental). In each table entry, the values of the parameters are such that the result lies on the axis between two quadrants; the corresponding accuracy rule, which takes precedence over the maximum relative error bounds, is that the result interval is the model interval of *EF*.Float_Type associated with the exact mathematical result given in the table.

12/1 *This paragraph was deleted.*

13 The last line of the table is meant to apply when *EF*.Float_Type'Signed_Zeros is False; the two lines just above it, when *EF*.Float_Type'Signed_Zeros is True and the parameter Y has a zero value with the indicated sign.

Table G-1: Tightly Approximated Elementary Function Results

Function	Value of X	Value of Y	Exact Result when Cycle Specified	Exact Result when Cycle Omitted
Arcsin	1.0	n.a.	Cycle/4.0	π/2.0
Arcsin	–1.0	n.a.	–Cycle/4.0	–π/2.0
Arccos	0.0	n.a.	Cycle/4.0	π/2.0
Arccos	–1.0	n.a.	Cycle/2.0	π
Arctan and Arccot	0.0	positive	Cycle/4.0	π/2.0
Arctan and Arccot	0.0	negative	–Cycle/4.0	–π/2.0
Arctan and Arccot	negative	+0.0	Cycle/2.0	π
Arctan and Arccot	negative	–0.0	–Cycle/2.0	–π
Arctan and Arccot	negative	0.0	Cycle/2.0	π

14 The amount by which the result of an inverse trigonometric function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in A.5.1, is limited. The rule is that the result belongs to the smallest model interval of *EF*.Float_Type that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence over the maximum relative error bounds, effectively narrowing the result interval allowed by them.

15 Finally, the following specifications also take precedence over the maximum relative error bounds:

16 • The absolute value of the result of the Sin, Cos, and Tanh functions never exceeds one.

17 • The absolute value of the result of the Coth function is never less than one.

18 • The result of the Cosh function is never less than one.

Implementation Advice

19 The versions of the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of 2.0*Numerics.Pi, since this will not provide the required accuracy in some portions of the domain. For the same reason, the version of Log without a Base parameter should not be implemented by calling the corresponding version with a Base parameter of Numerics.e.

19.a.1/2 **Implementation Advice:** For elementary functions, the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter. Log without a Base parameter should not be implemented by calling Log with a Base parameter.

Wording Changes from Ada 83

19.a The semantics of Numerics.Generic_Elementary_Functions differs from Generic_Elementary_Functions as defined in ISO/IEC DIS 11430 (for Ada 83) in the following ways related to the accuracy specified for strict mode:

19.b • The maximum relative error bounds use the Model_Epsilon attribute instead of the Base'Epsilon attribute.

- The accuracy requirements are expressed in terms of result intervals that are model intervals. On the one hand, this facilitates the description of the required results in the presence of underflow; on the other hand, it slightly relaxes the requirements expressed in ISO/IEC DIS 11430. 19.c

G.2.5 Performance Requirements for Random Number Generation

1 In the strict mode, the performance of Numerics.Float_Random and Numerics.Discrete_Random shall be as specified here.

Implementation Requirements

2 Two different calls to the time-dependent Reset procedure shall reset the generator to different states, provided that the calls are separated in time by at least one second and not more than fifty years.

3 The implementation's representations of generator states and its algorithms for generating random numbers shall yield a period of at least 2^{31}–2; much longer periods are desirable but not required.

4 The implementations of Numerics.Float_Random.Random and Numerics.Discrete_Random.Random shall pass at least 85% of the individual trials in a suite of statistical tests. For Numerics.Float_Random, the tests are applied directly to the floating point values generated (i.e., they are not converted to integers first), while for Numerics.Discrete_Random they are applied to the generated values of various discrete types. Each test suite performs 6 different tests, with each test repeated 10 times, yielding a total of 60 individual trials. An individual trial is deemed to pass if the chi-square value (or other statistic) calculated for the observed counts or distribution falls within the range of values corresponding to the 2.5 and 97.5 percentage points for the relevant degrees of freedom (i.e., it shall be neither too high nor too low). For the purpose of determining the degrees of freedom, measurement categories are combined whenever the expected counts are fewer than 5.

4.a **Implementation Note:** In the floating point random number test suite, the generator is reset to a time-dependent state at the beginning of the run. The test suite incorporates the following tests, adapted from D. E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms.* In the descriptions below, the given number of degrees of freedom is the number before reduction due to any necessary combination of measurement categories with small expected counts; it is one less than the number of measurement categories.

- Proportional Distribution Test (a variant of the Equidistribution Test). The interval 0.0 .. 1.0 is partitioned into K subintervals. K is chosen randomly between 4 and 25 for each repetition of the test, along with the boundaries of the subintervals (subject to the constraint that at least 2 of the subintervals have a width of 0.001 or more). 5000 random floating point numbers are generated. The counts of random numbers falling into each subinterval are tallied and compared with the expected counts, which are proportional to the widths of the subintervals. The number of degrees of freedom for the chi-square test is K–1. 4.b
- Gap Test. The bounds of a range A .. B, with $0.0 \le A < B \le 1.0$, are chosen randomly for each repetition of the test, subject to the constraint that $0.2 \le B–A \le 0.6$. Random floating point numbers are generated until 5000 falling into the range A .. B have been encountered. Each of these 5000 is preceded by a “gap” (of length greater than or equal to 0) of consecutive random numbers not falling into the range A .. B. The counts of gaps of each length from 0 to 15, and of all lengths greater than 15 lumped together, are tallied and compared with the expected counts. Let $P = B–A$. The probability that a gap has a length of L is $(1–P)^{L} \cdot P$ for $L \le 15$, while the probability that a gap has a length of 16 or more is $(1–P)^{16}$. The number of degrees of freedom for the chi-square test is 16. 4.c
- Permutation Test. 5000 tuples of 4 different random floating point numbers are generated. (An entire 4-tuple is discarded in the unlikely event that it contains any two exactly equal components.) The counts of each of the 4! = 24 possible relative orderings of the components of the 4-tuples are tallied and compared with the expected counts. Each of the possible relative orderings has an equal probability. The number of degrees of freedom for the chi-square test is 23. 4.d
- Increasing-Runs Test. Random floating point numbers are generated until 5000 increasing runs have been observed. An “increasing run” is a sequence of random numbers in strictly increasing order; it is followed by a random number that is strictly smaller than the preceding random number. (A run under construction is entirely discarded in the unlikely event that one random number is followed immediately by an exactly equal random number.) The decreasing random number that follows an increasing run is discarded and not included with the next increasing run. The counts of increasing runs of each length from 1 to 4, and of all lengths greater than 4 lumped together, are tallied and compared with the expected counts. The probability 4.e

that an increasing run has a length of L is $1/L! - 1/(L+1)!$ for $L \le 4$, while the probability that an increasing run has a length of 5 or more is 1/5!. The number of degrees of freedom for the chi-square test is 4.

4.f
- Decreasing-Runs Test. The test is similar to the Increasing Runs Test, but with decreasing runs.

4.g
- Maximum-of-*t* Test (with $t = 5$). 5000 tuples of 5 random floating point numbers are generated. The maximum of the components of each 5-tuple is determined and raised to the 5th power. The uniformity of the resulting values over the range 0.0 .. 1.0 is tested as in the Proportional Distribution Test.

4.h
Implementation Note: In the discrete random number test suite, Numerics.Discrete_Random is instantiated as described below. The generator is reset to a time-dependent state after each instantiation. The test suite incorporates the following tests, adapted from D. E. Knuth (*op. cit.*) and other sources. The given number of degrees of freedom for the chi-square test is reduced by any necessary combination of measurement categories with small expected counts, as described above.

4.i
- Equidistribution Test. In each repetition of the test, a number R between 2 and 30 is chosen randomly, and Numerics.Discrete_Random is instantiated with an integer subtype whose range is 1 .. R. 5000 integers are generated randomly from this range. The counts of occurrences of each integer in the range are tallied and compared with the expected counts, which have equal probabilities. The number of degrees of freedom for the chi-square test is R–1.

4.j
- Simplified Poker Test. Numerics.Discrete_Random is instantiated once with an enumeration subtype representing the 13 denominations (Two through Ten, Jack, Queen, King, and Ace) of an infinite deck of playing cards. 2000 “poker” hands (5-tuples of values of this subtype) are generated randomly. The counts of hands containing exactly K different denominations ($1 \le K \le 5$) are tallied and compared with the expected counts. The probability that a hand contains exactly K different denominations is given by a formula in Knuth. The number of degrees of freedom for the chi-square test is 4.

4.k
- Coupon Collector's Test. Numerics.Discrete_Random is instantiated in each repetition of the test with an integer subtype whose range is 1 .. R, where R varies systematically from 2 to 11. Integers are generated randomly from this range until each value in the range has occurred, and the number K of integers generated is recorded. This constitutes a “coupon collector's segment” of length K. 2000 such segments are generated. The counts of segments of each length from R to R+29, and of all lengths greater than R+29 lumped together, are tallied and compared with the expected counts. The probability that a segment has any given length is given by formulas in Knuth. The number of degrees of freedom for the chi-square test is 30.

4.l
- Craps Test (Lengths of Games). Numerics.Discrete_Random is instantiated once with an integer subtype whose range is 1 .. 6 (representing the six numbers on a die). 5000 craps games are played, and their lengths are recorded. (The length of a craps game is the number of rolls of the pair of dice required to produce a win or a loss. A game is won on the first roll if the dice show 7 or 11; it is lost if they show 2, 3, or 12. If the dice show some other sum on the first roll, it is called the *point*, and the game is won if and only if the point is rolled again before a 7 is rolled.) The counts of games of each length from 1 to 18, and of all lengths greater than 18 lumped together, are tallied and compared with the expected counts. For $2 \le S \le 12$, let D_S be the probability that a roll of a pair of dice shows the sum S, and let $Q_S(L) = D_S \cdot (1 - (D_S + D_7))^{L-2} \cdot (D_S + D_7)$. Then, the probability that a game has a length of 1 is $D_7 + D_{11} + D_2 + D_3 + D_{12}$ and, for $L > 1$, the probability that a game has a length of L is $Q_4(L) + Q_5(L) + Q_6(L) + Q_8(L) + Q_9(L) + Q_{10}(L)$. The number of degrees of freedom for the chi-square test is 18.

4.m
- Craps Test (Lengths of Passes). This test is similar to the last, but enough craps games are played for 3000 losses to occur. A string of wins followed by a loss is called a *pass*, and its length is the number of wins preceding the loss. The counts of passes of each length from 0 to 7, and of all lengths greater than 7 lumped together, are tallied and compared with the expected counts. For $L \ge 0$, the probability that a pass has a length of L is $W^L \cdot (1-W)$, where W, the probability that a game ends in a win, is 244.0/495.0. The number of degrees of freedom for the chi-square test is 8.

4.n
- Collision Test. Numerics.Discrete_Random is instantiated once with an integer or enumeration type representing binary bits. 15 successive calls on the Random function are used to obtain the bits of a 15-bit binary integer between 0 and 32767. 3000 such integers are generated, and the number of collisions (integers previously generated) is counted and compared with the expected count. A chi-square test is not used to assess the number of collisions; rather, the limits on the number of collisions, corresponding to the 2.5 and 97.5 percentage points, are (from formulas in Knuth) 112 and 154. The test passes if and only if the number of collisions is in this range.

G.2.6 Accuracy Requirements for Complex Arithmetic

1 In the strict mode, the performance of Numerics.Generic_Complex_Types and Numerics.Generic_-Complex_Elementary_Functions shall be as specified here.

Implementation Requirements

2 When an exception is not raised, the result of evaluating a real function of an instance *CT* of Numerics.Generic_Complex_Types (i.e., a function that yields a value of subtype *CT*.Real'Base or *CT*.Imaginary) belongs to a result interval defined as for a real elementary function (see G.2.4).

3 {*result interval (for a component of the result of evaluating a complex function)*} When an exception is not raised, each component of the result of evaluating a complex function of such an instance, or of an instance of Numerics.Generic_Complex_Elementary_Functions obtained by instantiating the latter with *CT* (i.e., a function that yields a value of subtype *CT*.Complex), also belongs to a *result interval*. The result intervals for the components of the result are either defined by a *maximum relative error* bound or by a *maximum box error* bound. {*maximum relative error (for a component of the result of evaluating a complex function)*} When the result interval for the real (resp., imaginary) component is defined by maximum relative error, it is defined as for that of a real function, relative to the exact value of the real (resp., imaginary) part of the result of the corresponding mathematical function. {*maximum box error (for a component of the result of evaluating a complex function)*} When defined by maximum box error, the result interval for a component of the result is the smallest model interval of *CT*.Real that contains all the values of the corresponding part of $f \cdot (1.0 + d)$, where f is the exact complex value of the corresponding mathematical function at the given parameter values, d is complex, and $|d|$ is less than or equal to the given maximum box error. {*Overflow_Check* [partial]} {*check, language-defined (Overflow_Check)*} The function delivers a value that belongs to the result interval (or a value both of whose components belong to their respective result intervals) when both bounds of the result interval(s) belong to the safe range of *CT*.Real; otherwise,

3.a **Discussion:** The maximum relative error could be specified separately for each component, but we do not take advantage of that freedom here.

3.b **Discussion:** Note that $f \cdot (1.0 + d)$ defines a small circular region of the complex plane centered at f, and the result intervals for the real and imaginary components of the result define a small rectangular box containing that circle.

3.c **Reason:** Box error is used when the computation of the result risks loss of significance in a component due to cancellation.

3.d **Ramification:** The components of a complex function that exhibits bounded relative error in each component have to have the correct sign. In contrast, one of the components of a complex function that exhibits bounded box error may have the wrong sign, since the dimensions of the box containing the result are proportional to the modulus of the mathematical result and not to either component of the mathematical result individually. Thus, for example, the box containing the computed result of a complex function whose mathematical result has a large modulus but lies very close to the imaginary axis might well straddle that axis, allowing the real component of the computed result to have the wrong sign. In this case, the distance between the computed result and the mathematical result is, nevertheless, a small fraction of the modulus of the mathematical result.

- 4 {*Constraint_Error (raised by failure of run-time check)*} if *CT*.Real'Machine_Overflows is True, the function either delivers a value that belongs to the result interval (or a value both of whose components belong to their respective result intervals) or raises Constraint_Error, signaling overflow;
- 5 if *CT*.Real'Machine_Overflows is False, the result is implementation defined.

5.a **Implementation defined:** The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the Machine_Overflows attribute of the corresponding real type is False.

6/2 {*AI95-00434-01*} The error bounds for particular complex functions are tabulated in table G-2. In the table, the error bound is given as the coefficient of *CT*.Real'Model_Epsilon.

7/1 *This paragraph was deleted.*

Table G-2: Error Bounds for Particular Complex Functions

Function or Operator	Nature of Result	Nature of Bound	Error Bound
Modulus	real	max. rel. error	3.0
Argument	real	max. rel. error	4.0
Compose_From_Polar	complex	max. rel. error	3.0
"*" (both operands complex)	complex	max. box error	5.0
"/" (right operand complex)	complex	max. box error	13.0
Sqrt	complex	max. rel. error	6.0
Log	complex	max. box error	13.0
Exp (complex parameter)	complex	max. rel. error	7.0
Exp (imaginary parameter)	complex	max. rel. error	2.0
Sin, Cos, Sinh, and Cosh	complex	max. rel. error	11.0
Tan, Cot, Tanh, and Coth	complex	max. rel. error	35.0
inverse trigonometric	complex	max. rel. error	14.0
inverse hyperbolic	complex	max. rel. error	14.0

8 The maximum relative error given above applies throughout the domain of the Compose_From_Polar function when the Cycle parameter is specified. When the Cycle parameter is omitted, the maximum relative error applies only when the absolute value of the parameter Argument is less than or equal to the angle threshold (see G.2.4). For the Exp function, and for the forward hyperbolic (resp., trigonometric) functions, the maximum relative error given above likewise applies only when the absolute value of the imaginary (resp., real) component of the parameter X (or the absolute value of the parameter itself, in the case of the Exp function with a parameter of pure-imaginary type) is less than or equal to the angle threshold. For larger angles, the accuracy is implementation defined.

8.a **Implementation defined:** The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold.

9 The prescribed results specified in G.1.2 for certain functions at particular parameter values take precedence over the error bounds; effectively, they narrow to a single value the result interval allowed by the error bounds for a component of the result. Additional rules with a similar effect are given below for certain inverse trigonometric and inverse hyperbolic functions, at particular parameter values for which a component of the mathematical result is transcendental. In each case, the accuracy rule, which takes precedence over the error bounds, is that the result interval for the stated result component is the model interval of *CT*.Real associated with the component's exact mathematical value. The cases in question are as follows:

10 • When the parameter X has the value zero, the real (resp., imaginary) component of the result of the Arccot (resp., Arccoth) function is in the model interval of *CT*.Real associated with the value π/2.0.

11 • When the parameter X has the value one, the real component of the result of the Arcsin function is in the model interval of *CT*.Real associated with the value π/2.0.

12 • When the parameter X has the value –1.0, the real component of the result of the Arcsin (resp., Arccos) function is in the model interval of *CT*.Real associated with the value –π/2.0 (resp., π).

12.a **Discussion:** It is possible to give many other prescribed results in which a component of the parameter is restricted to a similar model interval when the parameter X is appropriately restricted to an easily testable portion of the domain. We follow the proposed ISO/IEC standard for Generic_Complex_Elementary_Functions (for Ada 83) in not doing so, however.

{*AI95-00434-01*} The amount by which a component of the result of an inverse trigonometric or inverse hyperbolic function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in G.1.2, is limited. The rule is that the result belongs to the smallest model interval of *CT*.Real that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence over the maximum error bounds, effectively narrowing the result interval allowed by them. 13/2

Finally, the results allowed by the error bounds are narrowed by one further rule: The absolute value of each component of the result of the Exp function, for a pure-imaginary parameter, never exceeds one. 14

Implementation Advice

The version of the Compose_From_Polar function without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of 2.0*Numerics.Pi, since this will not provide the required accuracy in some portions of the domain. 15

Implementation Advice: For complex arithmetic, the Compose_From_Polar function without a Cycle parameter should not be implemented by calling Compose_From_Polar with a Cycle parameter. 15.a.1/2

Wording Changes from Ada 83

The semantics of Numerics.Generic_Complex_Types and Numerics.Generic_Complex_Elementary_Functions differs from Generic_Complex_Types and Generic_Complex_Elementary_Functions as defined in ISO/IEC CDs 13813 and 13814 (for Ada 83) in ways analogous to those identified for the elementary functions in G.2.4. In addition, we do not generally specify the signs of zero results (or result components), although those proposed standards do. 15.a

G.3 Vector and Matrix Manipulation

{*AI95-00296-01*} Types and operations for the manipulation of real vectors and matrices are provided in Generic_Real_Arrays, which is defined in G.3.1. Types and operations for the manipulation of complex vectors and matrices are provided in Generic_Complex_Arrays, which is defined in G.3.2. Both of these library units are generic children of the predefined package Numerics (see A.5). Nongeneric equivalents of these packages for each of the predefined floating point types are also provided as children of Numerics. 1/2

Discussion: Vector and matrix manipulation is defined in the Numerics Annex, rather than in the core, because it is considered to be a specialized need of (some) numeric applications. 1.a/2

These packages provide facilities that are similar to and replace those found in ISO/IEC 13813:1998 *Information technology — Programming languages — Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types)*. (The other facilities provided by that Standard were already provided in Ada 95.) In addition to the main facilities of that Standard, these packages also include subprograms for the solution of linear equations, matrix inversion, determinants, and the determination of the eigenvalues and eigenvectors of real symmetric matrices and Hermitian matrices. 1.b/2

Extensions to Ada 95

{*AI95-00296-01*} {*extensions to Ada 95*} This clause is new. It just provides an introduction to the following subclauses. 1.c/2

G.3.1 Real Vectors and Matrices

Static Semantics

1/2 {*AI95-00296-01*} {*AI95-00418-01*} The generic library package Numerics.Generic_Real_Arrays has the following declaration:

2/2
```
generic
   type Real is digits <>;
package Ada.Numerics.Generic_Real_Arrays is
   pragma Pure(Generic_Real_Arrays);
```

3/2
```
   -- Types
```

4/2
```
   type Real_Vector is array (Integer range <>) of Real'Base;
   type Real_Matrix is array (Integer range <>, Integer range <>)
                                                   of Real'Base;
```

5/2
```
   -- Subprograms for Real_Vector types
```

6/2
```
   -- Real_Vector arithmetic operations
```

7/2
```
   function "+"   (Right : Real_Vector)       return Real_Vector;
   function "-"   (Right : Real_Vector)       return Real_Vector;
   function "abs" (Right : Real_Vector)       return Real_Vector;
```

8/2
```
   function "+"   (Left, Right : Real_Vector) return Real_Vector;
   function "-"   (Left, Right : Real_Vector) return Real_Vector;
```

9/2
```
   function "*"   (Left, Right : Real_Vector) return Real'Base;
```

10/2
```
   function "abs" (Right : Real_Vector)       return Real'Base;
```

11/2
```
   -- Real_Vector scaling operations
```

12/2
```
   function "*" (Left : Real'Base;   Right : Real_Vector)
      return Real_Vector;
   function "*" (Left : Real_Vector; Right : Real'Base)
      return Real_Vector;
   function "/" (Left : Real_Vector; Right : Real'Base)
      return Real_Vector;
```

13/2
```
   -- Other Real_Vector operations
```

14/2
```
   function Unit_Vector (Index : Integer;
                         Order : Positive;
                         First : Integer := 1) return Real_Vector;
```

15/2
```
   -- Subprograms for Real_Matrix types
```

16/2
```
   -- Real_Matrix arithmetic operations
```

17/2
```
   function "+"       (Right : Real_Matrix) return Real_Matrix;
   function "-"       (Right : Real_Matrix) return Real_Matrix;
   function "abs"     (Right : Real_Matrix) return Real_Matrix;
   function Transpose (X     : Real_Matrix) return Real_Matrix;
```

18/2
```
   function "+" (Left, Right : Real_Matrix) return Real_Matrix;
   function "-" (Left, Right : Real_Matrix) return Real_Matrix;
   function "*" (Left, Right : Real_Matrix) return Real_Matrix;
```

19/2
```
   function "*" (Left, Right : Real_Vector) return Real_Matrix;
```

20/2
```
   function "*" (Left : Real_Vector; Right : Real_Matrix)
      return Real_Vector;
   function "*" (Left : Real_Matrix; Right : Real_Vector)
      return Real_Vector;
```

21/2
```
   -- Real_Matrix scaling operations
```

22/2
```
   function "*" (Left : Real'Base;   Right : Real_Matrix)
      return Real_Matrix;
   function "*" (Left : Real_Matrix; Right : Real'Base)
      return Real_Matrix;
   function "/" (Left : Real_Matrix; Right : Real'Base)
      return Real_Matrix;
      -- Real_Matrix inversion and related operations                                23/2

      function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;          24/2
      function Solve (A, X : Real_Matrix) return Real_Matrix;
      function Inverse (A : Real_Matrix) return Real_Matrix;
      function Determinant (A : Real_Matrix) return Real'Base;

      -- Eigenvalues and vectors of a real symmetric matrix                          25/2

      function Eigenvalues (A : Real_Matrix) return Real_Vector;                     26/2

      procedure Eigensystem (A       : in  Real_Matrix;                              27/2
                             Values  : out Real_Vector;
                             Vectors : out Real_Matrix);

      -- Other Real_Matrix operations                                                28/2

      function Unit_Matrix (Order            : Positive;                             29/2
                            First_1, First_2 : Integer := 1)
                                               return Real_Matrix;

   end Ada.Numerics.Generic_Real_Arrays;                                             30/2
```

{*AI95-00296-01*} The library package Numerics.Real_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Real_Arrays, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Real_Arrays, Numerics.Long_Real_Arrays, etc. 31/2

> **Reason:** The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics, and to be consistent with other Numerics packages. 31.a/2

{*AI95-00296-01*} Two types are defined and exported by Numerics.Generic_Real_Arrays. The composite type Real_Vector is provided to represent a vector with components of type Real; it is defined as an unconstrained, one-dimensional array with an index of type Integer. The composite type Real_Matrix is provided to represent a matrix with components of type Real; it is defined as an unconstrained, two-dimensional array with indices of type Integer. 32/2

{*AI95-00296-01*} The effect of the various subprograms is as described below. In most cases the subprograms are described in terms of corresponding scalar operations of the type Real; any exception raised by those operations is propagated by the array operation. Moreover, the accuracy of the result for each individual component is as defined for the scalar operation unless stated otherwise. 33/2

{*AI95-00296-01*} In the case of those operations which are defined to *involve an inner product*, Constraint_Error may be raised if an intermediate result is outside the range of Real'Base even though the mathematical final result would not be.{*involve an inner product (real)*} 34/2

```
   function "+"   (Right : Real_Vector) return Real_Vector;                          35/2
   function "-"   (Right : Real_Vector) return Real_Vector;
   function "abs" (Right : Real_Vector) return Real_Vector;
```

> {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index range of the result is Right'Range. 36/2

```
   function "+" (Left, Right : Real_Vector) return Real_Vector;                      37/2
   function "-" (Left, Right : Real_Vector) return Real_Vector;
```

> {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length. 38/2

```
   function "*" (Left, Right : Real_Vector) return Real'Base;                        39/2
```

> {*AI95-00296-01*} This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product. 40/2

41/2
```
function "abs" (Right : Real_Vector) return Real'Base;
```

42/2 {*AI95-00418-01*} This operation returns the L2-norm of Right (the square root of the inner product of the vector with itself).

42.a/2 **Discussion:** Normalization of vectors is a frequent enough operation that it is useful to provide the norm as a basic operation. Furthermore, implementing the norm is not entirely straightforward, because the inner product might overflow while the final norm does not. An implementation cannot merely return Sqrt (X * X), it has to cope with a possible overflow of the inner product.

42.b/2 **Implementation Note:** While the definition is given in terms of an inner product, the norm doesn't “involve an inner product” in the technical sense. The reason is that it has accuracy requirements substantially different from those applicable to inner products; and that cancellations cannot occur, because all the terms are positive, so there is no possibility of intermediate overflow.

43/2
```
function "*" (Left : Real'Base; Right : Real_Vector) return Real_Vector;
```

44/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index range of the result is Right'Range.

45/2
```
function "*" (Left : Real_Vector; Right : Real'Base) return Real_Vector;
function "/" (Left : Real_Vector; Right : Real'Base) return Real_Vector;
```

46/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index range of the result is Left'Range.

47/2
```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Real_Vector;
```

48/2 {*AI95-00296-01*} This function returns a *unit vector*{*unit vector (real vector)*} with Order components and a lower bound of First. All components are set to 0.0 except for the Index component which is set to 1.0. Constraint_Error is raised if Index < First, Index > First + Order – 1 or if First + Order – 1 > Integer'Last.

49/2
```
function "+"   (Right : Real_Matrix) return Real_Matrix;
function "-"   (Right : Real_Matrix) return Real_Matrix;
function "abs" (Right : Real_Matrix) return Real_Matrix;
```

50/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index ranges of the result are those of Right.

51/2
```
function Transpose (X : Real_Matrix) return Real_Matrix;
```

52/2 {*AI95-00296-01*} This function returns the transpose of a matrix X. The first and second index ranges of the result are X'Range(2) and X'Range(1) respectively.

53/2
```
function "+" (Left, Right : Real_Matrix) return Real_Matrix;
function "-" (Left, Right : Real_Matrix) return Real_Matrix;
```

54/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

55/2
```
function "*" (Left, Right : Real_Matrix) return Real_Matrix;
```

56/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). This operation involves inner products.

```
function "*" (Left, Right : Real_Vector) return Real_Matrix;
```
57/2

58/2 {*AI95-00296-01*} This operation returns the outer product of a (column) vector Left by a (row) vector Right using the operation "*" of the type Real for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

```
function "*" (Left : Real_Vector; Right : Real_Matrix) return Real_Vector;
```
59/2

60/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.

```
function "*" (Left : Real_Matrix; Right : Real_Vector) return Real_Vector;
```
61/2

62/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.

```
function "*" (Left : Real'Base; Right : Real_Matrix) return Real_Matrix;
```
63/2

64/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index ranges of the result are those of Right.

```
function "*" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;
function "/" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;
```
65/2

66/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index ranges of the result are those of Left.

```
function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;
```
67/2

68/2 {*AI95-00296-01*} This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

68.a/2 **Discussion:** The text says that Y is such that “X is (nearly) equal to A * Y” rather than “X is equal to A * Y” because rounding errors may mean that there is no value of Y such that X is exactly equal to A * Y. On the other hand it does not mean that any old rough value will do. The algorithm given under Implementation Advice should be followed.

68.b/2 The requirement to raise Constraint_Error if the matrix is ill-conditioned is really a reflection of what will happen if the matrix is ill-conditioned. See Implementation Advice. We do not make any attempt to define ill-conditioned formally.

68.c/2 These remarks apply to all versions of Solve and Inverse.

```
function Solve (A, X : Real_Matrix) return Real_Matrix;
```
69/2

70/2 {*AI95-00296-01*} This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Inverse (A : Real_Matrix) return Real_Matrix;
```
71/2

72/2 {*AI95-00296-01*} This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.

73/2
```
function Determinant (A : Real_Matrix) return Real'Base;
```

74/2 {*AI95-00296-01*} This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2).

75/2
```
function Eigenvalues(A : Real_Matrix) return Real_Vector;
```

76/2 {*AI95-00296-01*} This function returns the eigenvalues of the symmetric matrix A as a vector sorted into order with the largest first. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index range of the result is A'Range(1). Argument_Error is raised if the matrix A is not symmetric.

77/2
```
procedure Eigensystem(A       : in  Real_Matrix;
                      Values  : out Real_Vector;
                      Vectors : out Real_Matrix);
```

78/2 {*AI95-00296-01*} This procedure computes both the eigenvalues and eigenvectors of the symmetric matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are normalized and mutually orthogonal (they are orthonormal), including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index ranges of the parameter Vectors are those of A. Argument_Error is raised if the matrix A is not symmetric.

79/2
```
function Unit_Matrix (Order            : Positive;
                      First_1, First_2 : Integer := 1) return Real_Matrix;
```

80/2 {*AI95-00296-01*} This function returns a square *unit matrix*{*unit matrix (real matrix)*} with Order**2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to 0.0 except for the main diagonal, whose components are set to 1.0. Constraint_Error is raised if First_1 + Order – 1 > Integer'Last or First_2 + Order – 1 > Integer'Last.

Implementation Requirements

81/2 {*AI95-00296-01*} Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.

81.a/2 **Implementation defined:** The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Real_Matrix.

82/2 For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real in both the strict mode and the relaxed mode (see G.2).

83/2 For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product *X***Y* shall not exceed *g****abs**(*X*)***abs**(*Y*) where *g* is defined as

84/2 *g* = *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)

85/2 {*AI95-00418-01*} For the L2-norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed *g* / 2.0 + 3.0 * Real'Model_Epsilon where *g* is defined as above.

85.a/2 **Reason:** This is simply the combination of the error on the inner product with the error on Sqrt. A first order computation would lead to 2.0 * Real'Model_Epsilon above, but we are adding an extra Real'Model_Epsilon to account for higher order effects.

Documentation Requirements

{*AI95-00296-01*} Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic. 86/2

Documentation Requirement: Any techniques used to reduce cancellation errors in Numerics.Generic_Real_Arrays shall be documented. 86.a/2

Implementation Note: The above accuracy requirement is met by the canonical implementation of the inner product by multiplication and addition using the corresponding operations of type Real'Base and performing the cumulative addition using ascending indices. Note however, that some hardware provides special operations for the computation of the inner product and although these may be fast they may not meet the accuracy requirement specified. See Accuracy and Stability of Numerical Algorithms By N J Higham (ISBN 0-89871-355-2), Section 3.1. 86.b/2

Implementation Permissions

{*AI95-00296-01*} The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type. 87/2

Implementation Advice

{*AI95-00296-01*} Implementations should implement the Solve and Inverse functions using established techniques such as LU decomposition with row interchanges followed by back and forward substitution. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done then it should be documented. 88/2

Implementation Advice: Solve and Inverse for Numerics.Generic_Real_Arrays should be implemented using established techniques such as LU decomposition and the result should be refined by an iteration on the residuals. 88.a/2

It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise Constraint_Error is sufficient. 89/2

Discussion: There isn't any advice for the implementation to document with this paragraph. 89.a/2

The test that a matrix is symmetric should be performed by using the equality operator to compare the relevant components. 90/2

Implementation Advice: The equality operator should be used to test that a matrix in Numerics.Generic_Real_Matrix is symmetric. 90.a/2

Extensions to Ada 95

{*AI95-00296-01*} {*extensions to Ada 95*} The package Numerics.Generic_Real_Arrays and its nongeneric equivalents are new. 90.b/2

G.3.2 Complex Vectors and Matrices

Static Semantics

{*AI95-00296-01*} The generic library package Numerics.Generic_Complex_Arrays has the following declaration: 1/2

2/2
```
with Ada.Numerics.Generic_Real_Arrays, Ada.Numerics.Generic_Complex_Types;
generic
   with package Real_Arrays   is new
      Ada.Numerics.Generic_Real_Arrays   (<>);
   use Real_Arrays;
   with package Complex_Types is new
      Ada.Numerics.Generic_Complex_Types (Real);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Arrays is
   pragma Pure(Generic_Complex_Arrays);
```

3/2
```
   -- Types
```

4/2
```
   type Complex_Vector is array (Integer range <>) of Complex;
   type Complex_Matrix is array (Integer range <>,
                                 Integer range <>) of Complex;
```
5/2
```
   -- Subprograms for Complex_Vector types
```
6/2
```
   -- Complex_Vector selection, conversion and composition operations
```
7/2
```
   function Re (X : Complex_Vector) return Real_Vector;
   function Im (X : Complex_Vector) return Real_Vector;
```
8/2
```
   procedure Set_Re (X  : in out Complex_Vector;
                     Re : in     Real_Vector);
   procedure Set_Im (X  : in out Complex_Vector;
                     Im : in     Real_Vector);
```
9/2
```
   function Compose_From_Cartesian (Re     : Real_Vector)
      return Complex_Vector;
   function Compose_From_Cartesian (Re, Im : Real_Vector)
      return Complex_Vector;
```
10/2
```
   function Modulus  (X     : Complex_Vector) return Real_Vector;
   function "abs"    (Right : Complex_Vector) return Real_Vector
                                                 renames Modulus;
   function Argument (X     : Complex_Vector) return Real_Vector;
   function Argument (X     : Complex_Vector;
                      Cycle : Real'Base)      return Real_Vector;
```
11/2
```
   function Compose_From_Polar (Modulus, Argument : Real_Vector)
      return Complex_Vector;
   function Compose_From_Polar (Modulus, Argument : Real_Vector;
                                Cycle             : Real'Base)
      return Complex_Vector;
```
12/2
```
   -- Complex_Vector arithmetic operations
```
13/2
```
   function "+"       (Right  : Complex_Vector) return Complex_Vector;
   function "-"       (Right  : Complex_Vector) return Complex_Vector;
   function Conjugate (X      : Complex_Vector) return Complex_Vector;
```
14/2
```
   function "+"  (Left, Right : Complex_Vector) return Complex_Vector;
   function "-"  (Left, Right : Complex_Vector) return Complex_Vector;
```
15/2
```
   function "*"  (Left, Right : Complex_Vector) return Complex;
```
16/2
```
   function "abs"     (Right : Complex_Vector) return Complex;
```
17/2
```
   -- Mixed Real_Vector and Complex_Vector arithmetic operations
```
18/2
```
   function "+" (Left  : Real_Vector;
                 Right : Complex_Vector) return Complex_Vector;
   function "+" (Left  : Complex_Vector;
                 Right : Real_Vector)    return Complex_Vector;
   function "-" (Left  : Real_Vector;
                 Right : Complex_Vector) return Complex_Vector;
   function "-" (Left  : Complex_Vector;
                 Right : Real_Vector)    return Complex_Vector;
```
19/2
```
   function "*" (Left  : Real_Vector;    Right : Complex_Vector)
      return Complex;
   function "*" (Left  : Complex_Vector; Right : Real_Vector)
      return Complex;
```
20/2
```
   -- Complex_Vector scaling operations
```
21/2
```
   function "*" (Left  : Complex;
                 Right : Complex_Vector) return Complex_Vector;
   function "*" (Left  : Complex_Vector;
                 Right : Complex)        return Complex_Vector;
   function "/" (Left  : Complex_Vector;
                 Right : Complex)        return Complex_Vector;
```

22/2
```
function "*" (Left  : Real'Base;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real'Base)      return Complex_Vector;
function "/" (Left  : Complex_Vector;
              Right : Real'Base)      return Complex_Vector;
```

23/2
```
-- Other Complex_Vector operations
```

24/2
```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Complex_Vector;
```

25/2
```
-- Subprograms for Complex_Matrix types
```

26/2
```
-- Complex_Matrix selection, conversion and composition operations
```

27/2
```
function Re (X : Complex_Matrix) return Real_Matrix;
function Im (X : Complex_Matrix) return Real_Matrix;
```

28/2
```
procedure Set_Re (X  : in out Complex_Matrix;
                  Re : in     Real_Matrix);
procedure Set_Im (X  : in out Complex_Matrix;
                  Im : in     Real_Matrix);
```

29/2
```
function Compose_From_Cartesian (Re     : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix)
   return Complex_Matrix;
```

30/2
```
function Modulus  (X     : Complex_Matrix) return Real_Matrix;
function "abs"    (Right : Complex_Matrix) return Real_Matrix
                                              renames Modulus;
```

31/2
```
function Argument (X     : Complex_Matrix) return Real_Matrix;
function Argument (X     : Complex_Matrix;
                   Cycle : Real'Base)      return Real_Matrix;
```

32/2
```
function Compose_From_Polar (Modulus, Argument : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
   return Complex_Matrix;
```

33/2
```
-- Complex_Matrix arithmetic operations
```

34/2
```
function "+"       (Right : Complex_Matrix) return Complex_Matrix;
function "-"       (Right : Complex_Matrix) return Complex_Matrix;
function Conjugate (X     : Complex_Matrix) return Complex_Matrix;
function Transpose (X     : Complex_Matrix) return Complex_Matrix;
```

35/2
```
function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

36/2
```
function "*" (Left, Right : Complex_Vector) return Complex_Matrix;
```

37/2
```
function "*" (Left  : Complex_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;
```

38/2
```
-- Mixed Real_Matrix and Complex_Matrix arithmetic operations
```

39/2
```
function "+" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```

```
40/2       function "*" (Left  : Real_Vector;
                         Right : Complex_Vector) return Complex_Matrix;
           function "*" (Left  : Complex_Vector;
                         Right : Real_Vector)    return Complex_Matrix;
41/2       function "*" (Left  : Real_Vector;
                         Right : Complex_Matrix) return Complex_Vector;
           function "*" (Left  : Complex_Vector;
                         Right : Real_Matrix)    return Complex_Vector;
           function "*" (Left  : Real_Matrix;
                         Right : Complex_Vector) return Complex_Vector;
           function "*" (Left  : Complex_Matrix;
                         Right : Real_Vector)    return Complex_Vector;
42/2       -- Complex_Matrix scaling operations
43/2       function "*" (Left  : Complex;
                         Right : Complex_Matrix) return Complex_Matrix;
           function "*" (Left  : Complex_Matrix;
                         Right : Complex)        return Complex_Matrix;
           function "/" (Left  : Complex_Matrix;
                         Right : Complex)        return Complex_Matrix;
44/2       function "*" (Left  : Real'Base;
                         Right : Complex_Matrix) return Complex_Matrix;
           function "*" (Left  : Complex_Matrix;
                         Right : Real'Base)      return Complex_Matrix;
           function "/" (Left  : Complex_Matrix;
                         Right : Real'Base)      return Complex_Matrix;
45/2       -- Complex_Matrix inversion and related operations
46/2       function Solve (A : Complex_Matrix; X : Complex_Vector)
              return Complex_Vector;
           function Solve (A, X : Complex_Matrix) return Complex_Matrix;
           function Inverse (A : Complex_Matrix) return Complex_Matrix;
           function Determinant (A : Complex_Matrix) return Complex;
47/2       -- Eigenvalues and vectors of a Hermitian matrix
48/2       function Eigenvalues(A : Complex_Matrix) return Real_Vector;
49/2       procedure Eigensystem(A       : in  Complex_Matrix;
                                 Values  : out Real_Vector;
                                 Vectors : out Complex_Matrix);
50/2       -- Other Complex_Matrix operations
51/2       function Unit_Matrix (Order            : Positive;
                                 First_1, First_2 : Integer := 1)
                                                    return Complex_Matrix;
52/2    end Ada.Numerics.Generic_Complex_Arrays;
```

53/2 {*AI95-00296-01*} The library package Numerics.Complex_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Complex_Arrays, except that the predefined type Float is systematically substituted for Real'Base, and the Real_Vector and Real_Matrix types exported by Numerics.Real_Arrays are systematically substituted for Real_Vector and Real_Matrix, and the Complex type exported by Numerics.Complex_Types is systematically substituted for Complex, throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Arrays, Numerics.Long_Complex_Arrays, etc.

54/2 {*AI95-00296-01*} Two types are defined and exported by Numerics.Generic_Complex_Arrays. The composite type Complex_Vector is provided to represent a vector with components of type Complex; it is defined as an unconstrained one-dimensional array with an index of type Integer. The composite type Complex_Matrix is provided to represent a matrix with components of type Complex; it is defined as an unconstrained, two-dimensional array with indices of type Integer.

55/2 {*AI95-00296-01*} The effect of the various subprograms is as described below. In many cases they are described in terms of corresponding scalar operations in Numerics.Generic_Complex_Types. Any exception raised by those operations is propagated by the array subprogram. Moreover, any constraints on the parameters and the accuracy of the result for each individual component are as defined for the scalar operation.

{*AI95-00296-01*} In the case of those operations which are defined to *involve an inner product*, Constraint_Error may be raised if an intermediate result has a component outside the range of Real'Base even though the final mathematical result would not.{*involve an inner product (complex)*} 56/2

57/2
```
function Re (X : Complex_Vector) return Real_Vector;
function Im (X : Complex_Vector) return Real_Vector;
```

{*AI95-00296-01*} Each function returns a vector of the specified Cartesian components of X. The index range of the result is X'Range. 58/2

59/2
```
procedure Set_Re (X  : in out Complex_Vector; Re : in Real_Vector);
procedure Set_Im (X  : in out Complex_Vector; Im : in Real_Vector);
```

{*AI95-00296-01*} Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length is not equal to Re'Length or Im'Length. 60/2

61/2
```
function Compose_From_Cartesian (Re     : Real_Vector)
   return Complex_Vector;
function Compose_From_Cartesian (Re, Im : Real_Vector)
   return Complex_Vector;
```

{*AI95-00296-01*} Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index range of the result is Re'Range. Constraint_Error is raised if Re'Length is not equal to Im'Length. 62/2

63/2
```
function Modulus  (X     : Complex_Vector) return Real_Vector;
function "abs"    (Right : Complex_Vector) return Real_Vector
                                              renames Modulus;
function Argument (X     : Complex_Vector) return Real_Vector;
function Argument (X     : Complex_Vector;
                   Cycle : Real'Base)      return Real_Vector;
```

{*AI95-00296-01*} Each function calculates and returns a vector of the specified polar components of X or Right using the corresponding function in numerics.generic_complex_types. The index range of the result is X'Range or Right'Range. 64/2

65/2
```
function Compose_From_Polar (Modulus, Argument : Real_Vector)
   return Complex_Vector;
function Compose_From_Polar (Modulus, Argument : Real_Vector;
                             Cycle             : Real'Base)
   return Complex_Vector;
```

{*AI95-00296-01*} Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of polar components using the corresponding function in numerics.generic_complex_types on matching components of Modulus and Argument. The index range of the result is Modulus'Range. Constraint_Error is raised if Modulus'Length is not equal to Argument'Length. 66/2

67/2
```
function "+" (Right : Complex_Vector) return Complex_Vector;
function "-" (Right : Complex_Vector) return Complex_Vector;
```

{*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Right. The index range of the result is Right'Range. 68/2

69/2
```
function Conjugate (X : Complex_Vector) return Complex_Vector;
```

70/2 {*AI95-00296-01*} This function returns the result of applying the appropriate function Conjugate in numerics.generic_complex_types to each component of X. The index range of the result is X'Range.

71/2
```
function "+" (Left, Right : Complex_Vector) return Complex_Vector;
function "-" (Left, Right : Complex_Vector) return Complex_Vector;
```

72/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

73/2
```
function "*" (Left, Right : Complex_Vector) return Complex;
```

74/2 {*AI95-00296-01*} This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product.

75/2
```
function "abs" (Right : Complex_Vector) return Complex;
```

76/2 {*AI95-00418-01*} This operation returns the Hermitian L2-norm of Right (the square root of the inner product of the vector with its conjugate).

76.a/2 **Implementation Note:** While the definition is given in terms of an inner product, the norm doesn't "involve an inner product" in the technical sense. The reason is that it has accuracy requirements substantially different from those applicable to inner products; and that cancellations cannot occur, because all the terms are positive, so there is no possibility of intermediate overflow.

77/2
```
function "+" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "+" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
function "-" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "-" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
```

78/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

79/2
```
function "*" (Left : Real_Vector;    Right : Complex_Vector) return Complex;
function "*" (Left : Complex_Vector; Right : Real_Vector)    return Complex;
```

80/2 {*AI95-00296-01*} Each operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. These operations involve an inner product.

81/2
```
function "*" (Left : Complex; Right : Complex_Vector) return Complex_Vector;
```

82/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in numerics.generic_complex_types. The index range of the result is Right'Range.

83/2
```
function "*" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
function "/" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
```

84/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the vector Left and the complex number Right. The index range of the result is Left'Range.

85/2
```
function "*" (Left : Real'Base;
              Right : Complex_Vector) return Complex_Vector;
```

86/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in numerics.generic_complex_types. The index range of the result is Right'Range.

87/2
```
function "*" (Left : Complex_Vector;
              Right : Real'Base) return Complex_Vector;
function "/" (Left : Complex_Vector;
              Right : Real'Base) return Complex_Vector;
```

88/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the vector Left and the real number Right. The index range of the result is Left'Range.

89/2
```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Complex_Vector;
```

90/2 {*AI95-00296-01*} This function returns a *unit vector*{*unit vector (complex vector)*} with Order components and a lower bound of First. All components are set to (0.0, 0.0) except for the Index component which is set to (1.0, 0.0). Constraint_Error is raised if Index < First, Index > First + Order – 1, or if First + Order – 1 > Integer'Last.

91/2
```
function Re (X : Complex_Matrix) return Real_Matrix;
function Im (X : Complex_Matrix) return Real_Matrix;
```

92/2 {*AI95-00296-01*} Each function returns a matrix of the specified Cartesian components of X. The index ranges of the result are those of X.

93/2
```
procedure Set_Re (X : in out Complex_Matrix; Re : in Real_Matrix);
procedure Set_Im (X : in out Complex_Matrix; Im : in Real_Matrix);
```

94/2 {*AI95-00296-01*} Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length(1) is not equal to Re'Length(1) or Im'Length(1) or if X'Length(2) is not equal to Re'Length(2) or Im'Length(2).

95/2
```
function Compose_From_Cartesian (Re     : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix)
   return Complex_Matrix;
```

96/2 {*AI95-00296-01*} Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index ranges of the result are those of Re. Constraint_Error is raised if Re'Length(1) is not equal to Im'Length(1) or Re'Length(2) is not equal to Im'Length(2).

97/2
```
function Modulus  (X     : Complex_Matrix) return Real_Matrix;
function "abs"    (Right : Complex_Matrix) return Real_Matrix
                                              renames Modulus;
function Argument (X     : Complex_Matrix) return Real_Matrix;
function Argument (X     : Complex_Matrix;
                   Cycle : Real'Base)      return Real_Matrix;
```

98/2 {*AI95-00296-01*} Each function calculates and returns a matrix of the specified polar components of X or Right using the corresponding function in numerics.generic_complex_types. The index ranges of the result are those of X or Right.

99/2
```
function Compose_From_Polar (Modulus, Argument : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
   return Complex_Matrix;
```

100/2 {*AI95-00296-01*} Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of polar components using the corresponding function in numerics.generic_complex_types on matching components of Modulus and Argument. The index ranges of the result are those of Modulus. Constraint_Error is raised if Modulus'Length(1) is not equal to Argument'Length(1) or Modulus'Length(2) is not equal to Argument'Length(2).

101/2
```
function "+" (Right : Complex_Matrix) return Complex_Matrix;
function "-" (Right : Complex_Matrix) return Complex_Matrix;
```

102/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Right. The index ranges of the result are those of Right.

103/2
```
function Conjugate (X : Complex_Matrix) return Complex_Matrix;
```

104/2 {*AI95-00296-01*} This function returns the result of applying the appropriate function Conjugate in numerics.generic_complex_types to each component of X. The index ranges of the result are those of X.

105/2
```
function Transpose (X : Complex_Matrix) return Complex_Matrix;
```

106/2 {*AI95-00296-01*} This function returns the transpose of a matrix X. The first and second index ranges of the result are X'Range(2) and X'Range(1) respectively.

107/2
```
function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

108/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

109/2
```
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

110/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). This operation involves inner products.

111/2
```
function "*" (Left, Right : Complex_Vector) return Complex_Matrix;
```

112/2 {*AI95-00296-01*} This operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in numerics.generic_complex_types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

113/2
```
function "*" (Left  : Complex_Vector;
              Right : Complex_Matrix) return Complex_Vector;
```

114/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.

115/2
```
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;
```

116/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.

117/2
```
function "+" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```

118/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

119/2
```
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```

120/2 {*AI95-00296-01*} Each operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). These operations involve inner products.

121/2
```
function "*" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Matrix;
function "*" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Matrix;
```

122/2 {*AI95-00296-01*} Each operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in numerics.generic_complex_types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

123/2
```
function "*" (Left  : Real_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real_Matrix)    return Complex_Vector;
```

124/2 {*AI95-00296-01*} Each operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). These operations involve inner products.

125/2
```
function "*" (Left  : Real_Matrix;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Real_Vector)    return Complex_Vector;
```

126/2 {*AI95-00296-01*} Each operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. These operations involve inner products.

127/2
```
function "*" (Left : Complex; Right : Complex_Matrix) return Complex_Matrix;
```

128/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in numerics.generic_complex_types. The index ranges of the result are those of Right.

129/2
```
function "*" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;
function "/" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;
```

130/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the matrix Left and the complex number Right. The index ranges of the result are those of Left.

131/2
```
function "*" (Left : Real'Base;
              Right : Complex_Matrix) return Complex_Matrix;
```

132/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in numerics.generic_complex_types. The index ranges of the result are those of Right.

133/2
```
function "*" (Left : Complex_Matrix;
              Right : Real'Base) return Complex_Matrix;
function "/" (Left : Complex_Matrix;
              Right : Real'Base) return Complex_Matrix;
```

134/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the matrix Left and the real number Right. The index ranges of the result are those of Left.

135/2
```
function Solve (A : Complex_Matrix; X : Complex_Vector) return Complex_Vector;
```

136/2 {*AI95-00296-01*} This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

136.a/2 **Discussion:** The text says that Y is such that “X is (nearly) equal to A * Y” rather than “X is equal to A * Y” because rounding errors may mean that there is no value of Y such that X is exactly equal to A * Y. On the other hand it does not mean that any old rough value will do. The algorithm given under Implementation Advice should be followed.

136.b/2 The requirement to raise Constraint_Error if the matrix is ill-conditioned is really a reflection of what will happen if the matrix is ill-conditioned. See Implementation Advice. We do not make any attempt to define ill-conditioned formally.

136.c/2 These remarks apply to all versions of Solve and Inverse.

137/2
```
function Solve (A, X : Complex_Matrix) return Complex_Matrix;
```

138/2 {*AI95-00296-01*} This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

139/2
```
function Inverse (A : Complex_Matrix) return Complex_Matrix;
```

140/2 {*AI95-00296-01*} This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.

141/2
```
function Determinant (A : Complex_Matrix) return Complex;
```

142/2 {*AI95-00296-01*} This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2).

```
function Eigenvalues(A : Complex_Matrix) return Real_Vector;
```
143/2

{*AI95-00296-01*} This function returns the eigenvalues of the Hermitian matrix A as a vector sorted into order with the largest first. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index range of the result is A'Range(1). Argument_Error is raised if the matrix A is not Hermitian. 144/2

Discussion: A Hermitian matrix is one whose transpose is equal to its complex conjugate. The eigenvalues of a Hermitian matrix are always real. We only support this case because algorithms for solving the general case are inherently unstable. 144.a/2

```
procedure Eigensystem(A       : in  Complex_Matrix;
                      Values  :  out Real_Vector;
                      Vectors :  out Complex_Matrix);
```
145/2

{*AI95-00296-01*} This procedure computes both the eigenvalues and eigenvectors of the Hermitian matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are mutually orthonormal, including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index ranges of the parameter Vectors are those of A. Argument_Error is raised if the matrix A is not Hermitian. 146/2

```
function Unit_Matrix (Order            : Positive;
                      First_1, First_2 : Integer := 1)
                                         return Complex_Matrix;
```
147/2

{*AI95-00296-01*} This function returns a square *unit matrix*{*unit matrix (complex matrix)*} with Order**2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to (0.0, 0.0) except for the main diagonal, whose components are set to (1.0, 0.0). Constraint_Error is raised if First_1 + Order – 1 > Integer'Last or First_2 + Order – 1 > Integer'Last. 148/2

Implementation Requirements

{*AI95-00296-01*} Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined. 149/2

Implementation defined: The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Complex_Matrix. 149.a/2

{*AI95-00296-01*} For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real'Base and Complex in both the strict mode and the relaxed mode (see G.2). 150/2

{*AI95-00296-01*} For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product *X***Y* shall not exceed *g****abs**(*X*)***abs**(*Y*) where *g* is defined as 151/2

g = *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)
for mixed complex and real operands 152/2

g = sqrt(2.0) * *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)
for two complex operands 153/2

{*AI95-00418-01*} For the L2-norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed *g* / 2.0 + 3.0 * Real'Model_Epsilon where *g* has the definition appropriate for two complex operands. 154/2

Documentation Requirements

155/2 {*AI95-00296-01*} Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

155.a/2 **Documentation Requirement:** Any techniques used to reduce cancellation errors in Numerics.Generic_Complex_Arrays shall be documented.

155.b/2 **Implementation Note:** The above accuracy requirement is met by the canonical implementation of the inner product by multiplication and addition using the corresponding operations of type Complex and performing the cumulative addition using ascending indices. Note however, that some hardware provides special operations for the computation of the inner product and although these may be fast they may not meet the accuracy requirement specified. See Accuracy and Stability of Numerical Algorithms by N J Higham (ISBN 0-89871-355-2), Sections 3.1 and 3.6.

Implementation Permissions

156/2 {*AI95-00296-01*} The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

157/2 {*AI95-00296-01*} Although many operations are defined in terms of operations from numerics.-generic_complex_types, they need not be implemented by calling those operations provided that the effect is the same.

Implementation Advice

158/2 {*AI95-00296-01*} Implementations should implement the Solve and Inverse functions using established techniques. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done then it should be documented.

158.a/2 **Implementation Advice:** Solve and Inverse for Numerics.Generic_Complex_Arrays should be implemented using established techniques and the result should be refined by an iteration on the residuals.

159/2 {*AI95-00296-01*} It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise Constraint_Error is sufficient.

159.a/2 **Discussion:** There isn't any advice for the implementation to document with this paragraph.

160/2 {*AI95-00296-01*} The test that a matrix is Hermitian should use the equality operator to compare the real components and negation followed by equality to compare the imaginary components (see G.2.1).

160.a/2 **Implementation Advice:** The equality and negation operators should be used to test that a matrix is Hermitian.

161/2 {*AI95-00296-01*} Implementations should not perform operations on mixed complex and real operands by first converting the real operand to complex. See G.1.1.

161.a/2 **Implementation Advice:** Mixed real and complex operations should not be performed by converting the real operand to complex.

Extensions to Ada 95

161.b/2 {*AI95-00296-01*} {*extensions to Ada 95*} The package Numerics.Generic_Complex_Arrays and its nongeneric equivalents are new.

Annex H (normative) High Integrity Systems

{*AI95-00347-01*} {*safety-critical systems*} {*secure systems*} This Annex addresses requirements for high integrity systems (including safety-critical systems and security-critical systems). It provides facilities and specifies documentation requirements that relate to several needs: 1/2

- Understanding program execution; 2
- Reviewing object code; 3
- Restricting language constructs whose usage might complicate the demonstration of program correctness 4

Execution understandability is supported by pragma Normalize_Scalars, and also by requirements for the implementation to document the effect of a program in the presence of a bounded error or where the language rules leave the effect unspecified. {*unspecified* [partial]} 4.1

The pragmas Reviewable and Restrictions relate to the other requirements addressed by this Annex. 5

NOTES

1 The Valid attribute (see 13.9.2) is also useful in addressing these needs, to avoid problems that could otherwise arise from scalars that have values outside their declared range constraints. 6

Discussion: The Annex tries to provide high assurance rather than language features. However, it is not possible, in general, to test for high assurance. For any specific language feature, it is possible to demonstrate its presence by a functional test, as in the ACVC. One can also check for the presence of some documentation requirements, but it is not easy to determine objectively that the documentation is “adequate”. 6.a

Extensions to Ada 83

{*extensions to Ada 83*} This Annex is new to Ada 95. 6.b

Wording Changes from Ada 95

{*AI95-00347-01*} The title of this annex was changed to better reflect its purpose and scope. High integrity systems has become the standard way of identifying systems that have high reliability requirements; it subsumes terms such as safety and security. Moreover, the annex does not include any security specific features and as such the previous title is somewhat misleading. 6.c/2

H.1 Pragma Normalize_Scalars

This pragma ensures that an otherwise uninitialized scalar object is set to a predictable value, but out of range if possible. 1

Discussion: The goal of the pragma is to reduce the impact of a bounded error that results from a reference to an uninitialized scalar object, by having such a reference violate a range check and thus raise Constraint_Error. 1.a

Syntax

The form of a pragma Normalize_Scalars is as follows: 2

pragma Normalize_Scalars; 3

Post-Compilation Rules

{*configuration pragma (Normalize_Scalars)* [partial]} {*pragma, configuration (Normalize_Scalars)* [partial]} Pragma Normalize_Scalars is a configuration pragma. It applies to all compilation_units included in a partition. 4

Documentation Requirements

5/2 {*AI95-00434-01*} If a pragma Normalize_Scalars applies, the implementation shall document the implicit initial values for scalar subtypes, and shall identify each case in which such a value is used and is not an invalid representation.

5.a/2 **Documentation Requirement:** If a pragma Normalize_Scalars applies, the implicit initial values of scalar subtypes shall be documented. Such a value should be an invalid representation when possible; any cases when is it not shall be documented.

5.b **To be honest:** It's slightly inaccurate to say that the value is a representation, but the point should be clear anyway.

5.c **Discussion:** By providing a type with a size specification so that spare bits are present, it is possible to force an implementation of Normalize_Scalars to use an out of range value. This can be tested for by ensuring that Constraint_Error is raised. Similarly, for an unconstrained integer type, in which no spare bit is surely present, one can check that the initialization takes place to the value specified in the documentation of the implementation. For a floating point type, spare bits might not be available, but a range constraint can provide the ability to use an out of range value.

5.d If it is difficult to document the general rule for the implicit initial value, the implementation might choose instead to record the value on the object code listing or similar output produced during compilation.

Implementation Advice

6/2 {*AI95-00434-01*} Whenever possible, the implicit initial values for a scalar subtype should be an invalid representation (see 13.9.1).

6.a **Discussion:** When an out of range value is used for the initialization, it is likely that constraint checks will detect it. In addition, it can be detected by the Valid attribute.

6.b/2 This rule is included in the documentation requirements, and thus does not need a separate summary item.

NOTES

7 2 The initialization requirement applies to uninitialized scalar objects that are subcomponents of composite objects, to allocated objects, and to stand-alone objects. It also applies to scalar **out** parameters. Scalar subcomponents of composite **out** parameters are initialized to the corresponding part of the actual, by virtue of 6.4.1.

8 3 The initialization requirement does not apply to a scalar for which pragma Import has been specified, since initialization of an imported object is performed solely by the foreign language environment (see B.1).

9 4 The use of pragma Normalize_Scalars in conjunction with Pragma Restrictions(No_Exceptions) may result in erroneous execution (see H.4).

9.a **Discussion:** Since the effect of an access to an out of range value will often be to raise Constraint_Error, it is clear that suppressing the exception mechanism could result in erroneous execution. In particular, the assignment to an array, with the array index out of range, will result in a write to an arbitrary store location, having unpredictable effects.

H.2 Documentation of Implementation Decisions

Documentation Requirements

1 {*unspecified* [partial]} The implementation shall document the range of effects for each situation that the language rules identify as either a bounded error or as having an unspecified effect. If the implementation can constrain the effects of erroneous execution for a given construct, then it shall document such constraints. [The documentation might be provided either independently of any compilation unit or partition, or as part of an annotated listing for a given unit or partition. See also 1.1.3, and 1.1.2.]

1.a/2 *This paragraph was deleted.*

1.b/2 **Documentation Requirement:** The range of effects for each bounded error and each unspecified effect. If the effects of a given erroneous construct are constrained, the constraints shall be documented.

NOTES

2 5 Among the situations to be documented are the conventions chosen for parameter passing, the methods used for the management of run-time storage, and the method used to evaluate numeric expressions if this involves extended range or extra precision.

2.a **Discussion:** Look up “unspecified” and “erroneous execution” in the index for a list of the cases.

The management of run-time storage is particularly important. For safety applications, it is often necessary to show that a program cannot raise Storage_Error, and for security applications that information cannot leak via the run-time system. Users are likely to prefer a simple storage model that can be easily validated. 2.b

The documentation could helpfully take into account that users may well adopt a subset to avoid some forms of erroneous execution, for instance, not using the abort statement, so that the effects of a partly completed assignment_statement do not have to be considered in the validation of a program (see 9.8). For this reason documentation linked to an actual compilation may be most useful. Similarly, an implementation may be able to take into account use of the Restrictions pragma. 2.c

H.3 Reviewable Object Code

Object code review and validation are supported by pragmas Reviewable and Inspection_Point. 1

H.3.1 Pragma Reviewable

This pragma directs the implementation to provide information to facilitate analysis and review of a program's object code, in particular to allow determination of execution time and storage usage and to identify the correspondence between the source and object programs. 1

Discussion: Since the purpose of this pragma is to provide information to the user, it is hard to objectively test for conformity. In practice, users want the information in an easily understood and convenient form, but neither of these properties can be easily measured. 1.a

Syntax

The form of a pragma Reviewable is as follows: 2

pragma Reviewable; 3

Post-Compilation Rules

{*configuration pragma (Reviewable)* [partial]} {*pragma, configuration (Reviewable)* [partial]} Pragma Reviewable is a configuration pragma. It applies to all compilation_units included in a partition. 4

Implementation Requirements

The implementation shall provide the following information for any compilation unit to which such a pragma applies: 5

Discussion: The list of requirements can be checked for, even if issues like intelligibility are not addressed. 5.a

- Where compiler-generated run-time checks remain; 6

Discussion: A constraint check which is implemented via a check on the upper and lower bound should clearly be indicated. If a check is implicit in the form of machine instructions used (such an overflow checking), this should also be covered by the documentation. It is particularly important to cover those checks which are not obvious from the source code, such as that for stack overflow. 6.a

- An identification of any construct with a language-defined check that is recognized prior to run time as certain to fail if executed (even if the generation of run-time checks has been suppressed); 7

Discussion: In this case, if the compiler determines that a check must fail, the user should be informed of this. However, since it is not in general possible to know what the compiler will detect, it is not easy to test for this. In practice, it is thought that compilers claiming conformity to this Annex will perform significant optimizations and therefore *will* detect such situations. Of course, such events could well indicate a programmer error. 7.a

- {*AI95-00209-01*} For each read of a scalar object, an identification of the read as either “known to be initialized,” or “possibly uninitialized,” independent of whether pragma Normalize_Scalars applies; 8/2

Discussion: This issue again raises the question as to what the compiler has determined. A lazy implementation could clearly mark all scalars as “possibly uninitialized”, but this would be very unhelpful to the user. It should be possible to analyze a range of scalar uses and note the percentage in each class. Note that an access marked “known to be 8.a initialized” does not imply that the value is in range, since the initialization could be from an (erroneous) call of unchecked conversion, or by means external to the Ada program.

9 • Where run-time support routines are implicitly invoked;

9.a **Discussion:** Validators will need to know the calls invoked in order to check for the correct functionality. For instance, for some safety applications, it may be necessary to ensure that certain sections of code can execute in a particular time.

10 • An object code listing, including:

11 • Machine instructions, with relative offsets;

11.a **Discussion:** The machine instructions should be in a format that is easily understood, such as the symbolic format of the assembler. The relative offsets are needed in numeric format, to check any alignment restrictions that the architecture might impose.

12 • Where each data object is stored during its lifetime;

12.a **Discussion:** This requirement implies that if the optimizer assigns a variable to a register, this needs to be evident.

13 • Correspondence with the source program, including an identification of the code produced per declaration and per statement.

13.a **Discussion:** This correspondence will be quite complex when extensive optimization is performed. In particular, address calculation to access some data structures could be moved from the actual access. However, when all the machine code arising from a statement or declaration is in one basic block, this must be indicated by the implementation.

14 • An identification of each construct for which the implementation detects the possibility of erroneous execution;

14.a **Discussion:** This requirement is quite vague. In general, it is hard for compilers to detect erroneous execution and therefore the requirement will be rarely invoked. However, if the pragma Suppress is used and the compiler can show that a predefined exception will be raised, then such an identification would be useful.

15 • For each subprogram, block, task, or other construct implemented by reserving and subsequently freeing an area on a run-time stack, an identification of the length of the fixed-size portion of the area and an indication of whether the non-fixed size portion is reserved on the stack or in a dynamically-managed storage region.

15.a **Discussion:** This requirement is vital for those requiring to show that the storage available to a program is sufficient. This is crucial in those cases in which the internal checks for stack overflow are suppressed (perhaps by **pragma** Restrictions(No_Exceptions)).

16 The implementation shall provide the following information for any partition to which the pragma applies:

17 • An object code listing of the entire partition, including initialization and finalization code as well as run-time system components, and with an identification of those instructions and data that will be relocated at load time;

17.a **Discussion:** The object code listing should enable a validator to estimate upper bounds for the time taken by critical parts of a program. Similarly, by an analysis of the entire partition, it should be possible to ensure that the storage requirements are suitably bounded, assuming that the partition was written in an appropriate manner.

18 • A description of the run-time model relevant to the partition.

18.a **Discussion:** For example, a description of the storage model is vital, since the Ada language does not explicitly define such a model.

18.1 The implementation shall provide control- and data-flow information, both within each compilation unit and across the compilation units of the partition.

18.b **Discussion:** This requirement is quite vague, since it is unclear what control and data flow information the compiler has produced. It is really a plea not to throw away information that could be useful to the validator. Note that the data flow information is relevant to the detection of “possibly uninitialized” objects referred to above.

Implementation Advice

The implementation should provide the above information in both a human-readable and machine-readable form, and should document the latter so as to ease further processing by automated tools. 19

Implementation Advice: The information produced by pragma Reviewable should be provided in both a human-readable and machine-readable form, and the latter form should be documented. 19.a/2

Object code listings should be provided both in a symbolic format and also in an appropriate numeric format (such as hexadecimal or octal). 20

Implementation Advice: Object code listings should be provided both in a symbolic format and in a numeric format. 20.a/2

Reason: This is to enable other tools to perform any analysis that the user needed to aid validation. The format should be in some agreed form. 20.b

NOTES

6 The order of elaboration of library units will be documented even in the absence of pragma Reviewable (see 10.2). 21

Discussion: There might be some interactions between pragma Reviewable and compiler optimizations. For example, an implementation may disable some optimizations when pragma Reviewable is in force if it would be overly complicated to provide the detailed information to allow review of the optimized object code. See also pragma Optimize (2.8). 21.a

Wording Changes from Ada 95

{*AI95-00209-01*} The wording was clarified that pragma Reviewable applies to each read of an object, as it makes no sense to talk about the state of an object that will immediately be overwritten. 21.b/2

H.3.2 Pragma Inspection_Point

An occurrence of a pragma Inspection_Point identifies a set of objects each of whose values is to be available at the point(s) during program execution corresponding to the position of the pragma in the compilation unit. The purpose of such a pragma is to facilitate code validation. 1

Discussion: Inspection points are a high level equivalent of break points used by debuggers. 1.a

Syntax

The form of a pragma Inspection_Point is as follows: 2

pragma Inspection_Point[(*object*_name {, *object*_name})]; 3

Legality Rules

A pragma Inspection_Point is allowed wherever a declarative_item or statement is allowed. Each *object*_name shall statically denote the declaration of an object. 4

Discussion: The static denotation is required, since no dynamic evaluation of a name is involved in this pragma. 4.a

Static Semantics

{*8652/0093*} {*AI95-00207-01*} {*AI95-00434-01*} {*inspection point*} An *inspection point* is a point in the object code corresponding to the occurrence of a pragma Inspection_Point in the compilation unit. {*inspectable object*} An object is *inspectable* at an inspection point if the corresponding pragma Inspection_Point either has an argument denoting that object, or has no arguments and the declaration of the object is visible at the inspection point. 5/2

Ramification: If a pragma Inspection_Point is in an in-lined subprogram, there might be numerous inspection points in the object code corresponding to the single occurrence of the pragma in the source; similar considerations apply if such a pragma is in a generic, or in a loop that has been “unrolled” by an optimizer. 5.a

{*8652/0093*} {*AI95-00207-01*} The short form of the pragma is a convenient shorthand for listing all objects which could be explicitly made inspectable by the long form of the pragma; thus only visible objects are made inspectable by it. Objects that are not visible at the point of the pragma are not made inspectable by the short form pragma. This is 5.a.1/1 necessary so that implementations need not keep information about (or prevent optimizations on) a unit simply because some other unit *might* contain a short form Inspection_Point pragma.

5.b/1 **Discussion:** {*8652/0093*} {*AI95-00207-01*} If the short form of the pragma is used, then all visible objects are inspectable. This implies that global objects from other compilation units are inspectable. A good interactive debugging system could provide information similar to a post-mortem dump at such inspection points. The annex does not require that any inspection facility is provided, merely that the information is available to understand the state of the machine at those points.

Dynamic Semantics

6 Execution of a pragma Inspection_Point has no effect.

6.a/2 **Discussion:** {*AI95-00114-01*} Although an inspection point has no (semantic) effect, the removal or adding of a new point could change the machine code generated by the compiler.

Implementation Requirements

7 Reaching an inspection point is an external interaction with respect to the values of the inspectable objects at that point (see 1.1.3).

7.a **Ramification:** The compiler is inhibited from moving an assignment to an inspectable variable past an inspection point for that variable. On the other hand, the evaluation of an expression that might raise an exception may be moved past an inspection point (see 11.6).

Documentation Requirements

8 For each inspection point, the implementation shall identify a mapping between each inspectable object and the machine resources (such as memory locations or registers) from which the object's value can be obtained.

8.a/2 *This paragraph was deleted.*

8.b/2 **Documentation Requirement:** For each inspection point, a mapping between each inspectable object and the machine resources where the object's value can be obtained shall be provided.

NOTES

9/2 7 {*AI95-00209-01*} The implementation is not allowed to perform “dead store elimination” on the last assignment to a variable prior to a point where the variable is inspectable. Thus an inspection point has the effect of an implicit read of each of its inspectable objects.

10 8 Inspection points are useful in maintaining a correspondence between the state of the program in source code terms, and the machine state during the program's execution. Assertions about the values of program objects can be tested in machine terms at inspection points. Object code between inspection points can be processed by automated tools to verify programs mechanically.

10.a **Discussion:** Although it is not a requirement of the annex, it would be useful if the state of the stack and heap could be interrogated. This would allow users to check that a program did not have a `storage leak'.

11 9 The identification of the mapping from source program objects to machine resources is allowed to be in the form of an annotated object listing, in human-readable or tool-processable form.

11.a **Discussion:** In principle, it is easy to check an implementation for this pragma, since one merely needs to check the content of objects against those values known from the source listing. In practice, one needs a tool similar to an interactive debugger to perform the check.

Wording Changes from Ada 95

11.b/2 {*8652/0093*} {*AI95-00207-01*} **Corrigendum:** Corrected the definition of the Inspection_Point pragma to apply to only variables visible at the point of the pragma. Otherwise, the compiler would have to assume that some other code somewhere could have a pragma Inspection_Point, preventing many optimizations (such as unused object elimination).

H.4 High Integrity Restrictions

1 This clause defines restrictions that can be used with pragma Restrictions (see 13.12); these facilitate the demonstration of program correctness by allowing tailored versions of the run-time system.

Discussion: Note that the restrictions are absolute. If a partition has 100 library units and just one needs Unchecked_Conversion, then the pragma cannot be used to ensure the other 99 units do not use Unchecked_Conversion. Note also that these are restrictions on all Ada code within a partition, and therefore it may not be evident from the specification of a package whether a restriction can be imposed. 1.a

Static Semantics

This paragraph was deleted.{*AI95-00347-01*} {*AI95-00394-01*} 2/2

{*AI95-00394-01*} The following *restriction*_identifiers are language defined: 3/2

Tasking-related restriction: 4

{*Restrictions (No_Protected_Types)*} No_Protected_Types There are no declarations of protected types or protected objects. 5

Memory-management related restrictions: 6

{*Restrictions (No_Allocators)*} No_Allocators There are no occurrences of an allocator. 7

{*8652/0042*} {*AI95-00130*} {*Restrictions (No_Local_Allocators)*} No_Local_Allocators Allocators are prohibited in subprograms, generic subprograms, tasks, and entry bodies. 8/1

Ramification: Thus allocators are permitted only in expressions whose evaluation can only be performed before the main subprogram is invoked. 8.a

This paragraph was deleted.{*8652/0042*} {*AI95-00130*} 8.b/1

This paragraph was deleted.{*AI95-00394-01*} 9/2

This paragraph was deleted. 9.a/2

Immediate_Reclamation 10
Except for storage occupied by objects created by allocators and not deallocated via unchecked deallocation, any storage reserved at run time for an object is immediately reclaimed when the object no longer exists. {*Restrictions (Immediate_Reclamation)*}

Discussion: Immediate reclamation would apply to storage created by the compiler, such as for a return value from a function whose size is not known at the call site. 10.a

Exception-related restriction: 11

{*Restrictions (No_Exceptions)*} No_Exceptions Raise_statements and exception_handlers are not allowed. No language-defined run-time checks are generated; however, a run-time check performed automatically by the hardware is permitted. 12

Discussion: This restriction mirrors a method of working that is quite common in the safety area. The programmer is required to show that exceptions cannot be raised. Then a simplified run-time system is used without exception handling. However, some hardware checks may still be enforced. If the software check would have failed, or if the hardware check actually fails, then the execution of the program is unpredictable. There are obvious dangers in this approach, but it is similar to programming at the assembler level. 12.a

Other restrictions: 13

{*Restrictions (No_Floating_Point)*} No_Floating_Point Uses of predefined floating point types and operations, and declarations of new floating point types, are not allowed. 14

Discussion: {*AI95-00114-01*} The intention is to avoid the use of floating point hardware at run time, but this is expressed in language terms. It is conceivable that floating point is used implicitly in some contexts, say fixed point type conversions of high accuracy. However, the Implementation Requirements below make it clear that the restriction would apply to the "run-time system" and hence not be allowed. This restriction could be used to inform a compiler that a variant of the architecture is being used which does not have floating point instructions. 14.a/2

{*Restrictions (No_Fixed_Point)*} No_Fixed_Point Uses of predefined fixed point types and operations, and declarations of new fixed point types, are not allowed. 15

Discussion: This restriction would have the side-effect of prohibiting the delay_relative_statement. As with the No_Floating_Point restriction, this might be used to avoid any question of rounding errors. Unless an Ada run-time is 15.a written in Ada, it seems hard to rule out implicit use of fixed point, since at the machine level, fixed point is virtually the same as integer arithmetic.

16/2 *This paragraph was deleted.*{*AI95-00394-01*}

16.a/2 *This paragraph was deleted.*

17 No_Access_Subprograms
The declaration of access-to-subprogram types is not allowed. {*Restrictions (No_Access_Subprograms)*}

17.a.1/2 **Discussion:** Most critical applications would require some restrictions or additional validation checks on uses of access-to-subprogram types. If the application does not require the functionality, then this restriction provides a means of ensuring the design requirement has been satisfied. The same applies to several of the following restrictions, and to restriction No_Dependence => Ada.Unchecked_Conversion.

18 {*Restrictions (No_Unchecked_Access)*} No_Unchecked_Access The Unchecked_Access attribute is not allowed.

19 {*Restrictions (No_Dispatch)*} No_Dispatch
Occurrences of T'Class are not allowed, for any (tagged) subtype T.

20/2 {*AI95-00285-01*} {*Restrictions (No_IO)*} No_IO
Semantic dependence on any of the library units Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, or Stream_IO is not allowed.

20.a **Discussion:** Excluding the input-output facilities of an implementation may be needed in those environments which cannot support the supplied functionality. A program in such an environment is likely to require some low level facilities or a call on a non-Ada feature.

21 {*Restrictions (No_Delay)*} No_Delay
Delay_Statements and semantic dependence on package Calendar are not allowed.

21.a **Ramification:** This implies that delay_alternatives in a select_statement are prohibited.

21.b The purpose of this restriction is to avoid the need for timing facilities within the run-time system.

22 {*Restrictions (No_Recursion)*} No_Recursion As part of the execution of a subprogram, the same subprogram is not invoked.

23 {*Restrictions (No_Reentrancy)*} No_Reentrancy During the execution of a subprogram by a task, no other task invokes the same subprogram.

Implementation Requirements

23.1/2 {*AI95-00394-01*} An implementation of this Annex shall support:

23.2/2 - the restrictions defined in this subclause; and

23.3/2 - the following restrictions defined in D.7: No_Task_Hierarchy, No_Abort_Statement, No_Implicit_Heap_Allocation; and

23.4/2 - {*AI95-00347-01*} the **pragma** Profile(Ravenscar); and

23.a/2 **Discussion:** {*AI95-00347-01*} The reference to pragma Profile(Ravenscar) is intended to show that properly restricted tasking is appropriate for use in high integrity systems. The Ada 95 Annex seemed to suggest that tasking was inappropriate for such systems.

23.5/2 - the following uses of *restriction_parameter*_identifiers defined in D.7[, which are checked prior to program execution]:

23.6/2
  - Max_Task_Entries => 0,

23.7/2
  - Max_Asynchronous_Select_Nesting => 0, and

23.8/2
  - Max_Tasks => 0.

24 If an implementation supports pragma Restrictions for a particular argument, then except for the restrictions No_Unchecked_Deallocation, No_Unchecked_Conversion, No_Access_Subprograms, and No_Unchecked_Access, the associated restriction applies to the run-time system.

24.a **Reason:** Permission is granted for the run-time system to use the specified otherwise-restricted features, since the use of these features may simplify the run-time system by allowing more of it to be written in Ada.

24.b **Discussion:** The restrictions that are applied to the partition are also applied to the run-time system. For example, if No_Floating_Point is specified, then an implementation that uses floating point for implementing the delay statement (say) would require that No_Floating_Point is only used in conjunction with No_Delay. It is clearly important that restrictions are effective so that Max_Tasks=0 does imply that tasking is not used, even implicitly (for input-output, say).

24.c An implementation of tasking could be produced based upon a run-time system written in Ada in which the rendezvous was controlled by protected types. In this case, No_Protected_Types could only be used in conjunction with Max_Task_Entries=0. Other implementation dependencies could be envisaged.

24.d If the run-time system is not written in Ada, then the wording needs to be applied in an appropriate fashion.

Documentation Requirements

25 If a pragma Restrictions(No_Exceptions) is specified, the implementation shall document the effects of all constructs where language-defined checks are still performed automatically (for example, an overflow check performed by the processor).

25.a/2 *This paragraph was deleted.*

25.b/2 **Documentation Requirement:** If a pragma Restrictions(No_Exceptions) is specified, the effects of all constructs where language-defined checks are still performed.

25.c/2 **Discussion:** {*AI95-00114-01*} The documentation requirements here are quite difficult to satisfy. One method is to review the object code generated and determine the checks that are still present, either explicitly, or implicitly within the architecture. As another example from that of overflow, consider the question of dereferencing a null pointer. This could be undertaken by a memory access trap when checks are performed. When checks are suppressed via the argument No_Exceptions, it would not be necessary to have the memory access trap mechanism enabled.

Erroneous Execution

26 {*erroneous execution (cause)* [partial]} Program execution is erroneous if pragma Restrictions(No_Exceptions) has been specified and the conditions arise under which a generated language-defined run-time check would fail.

26.a **Discussion:** The situation here is very similar to the application of pragma Suppress. Since users are removing some of the protection the language provides, they had better be careful!

27 {*erroneous execution (cause)* [partial]} Program execution is erroneous if pragma Restrictions(No_Recursion) has been specified and a subprogram is invoked as part of its own execution, or if pragma Restrictions(No_Reentrancy) has been specified and during the execution of a subprogram by a task, another task invokes the same subprogram.

27.a **Discussion:** In practice, many implementations may not exploit the absence of recursion or need for reentrancy, in which case the program execution would be unaffected by the use of recursion or reentrancy, even though the program is still formally erroneous.

27.b/2 *This paragraph was deleted.*

NOTES

28/2 10 {*AI95-00394-01*} Uses of *restriction_parameter_*identifier No_Dependence defined in 13.12.1: No_Dependence => Ada.Unchecked_Deallocation and No_Dependence => Ada.Unchecked_Conversion may be appropriate for high-integrity systems. Other uses of No_Dependence can also be appropriate for high-integrity systems.

28.a/2 **Discussion:** The specific mention of these two uses is meant to replace the identifiers now banished to J.13, “Dependence Restriction Identifiers”.

28.b/2 Restriction No_Dependence => Ada.Unchecked_Deallocation would be useful in those contexts in which heap storage is needed on program start-up, but need not be increased subsequently. The danger of a dangling pointer can therefore be avoided.

Extensions to Ada 95

28.c/2 {*8652/0042*} {*AI95-00130-01*} {*extensions to Ada 95*} No_Local_Allocators no longer prohibits generic instantiations.

Wording Changes from Ada 95

28.d/2 {*AI95-00285-01*} Wide_Wide_Text_IO (which is new) is added to the No_IO restriction.

28.e/2 {*AI95-00347-01*} The title of this clause was changed to match the change to the Annex title. Pragma Profile(Ravenscar) is part of this annex.

28.f/2 {*AI95-00394-01*} Restriction No_Dependence is used instead of special *restriction_*identifiers. The old names are banished to Obsolescent Features (see J.13).

28.g/2 {*AI95-00394-01*} The bizarre wording "apply in this Annex" (which no one quite can explain the meaning of) is banished.

H.5 Pragma Detect_Blocking

1/2 {*AI95-00305-01*} The following pragma forces an implementation to detect potentially blocking operations within a protected operation.

Syntax

2/2 {*AI95-00305-01*} The form of a pragma Detect_Blocking is as follows:

3/2 **pragma** Detect_Blocking;

Post-Compilation Rules

4/2 {*AI95-00305-01*} {*configuration pragma (Detect_Blocking)* [partial]} {*pragma, configuration (Detect_Blocking)* [partial]} A pragma Detect_Blocking is a configuration pragma.

Dynamic Semantics

5/2 {*AI95-00305-01*} An implementation is required to detect a potentially blocking operation within a protected operation, and to raise Program_Error (see 9.5.1).

Implementation Permissions

6/2 {*AI95-00305-01*} An implementation is allowed to reject a compilation_unit if a potentially blocking operation is present directly within an entry_body or the body of a protected subprogram.

NOTES

7/2 11 {*AI95-00305-01*} An operation that causes a task to be blocked within a foreign language domain is not defined to be potentially blocking, and need not be detected.

Extensions to Ada 95

7.a/2 {*AI95-00305-01*} {*extensions to Ada 95*} Pragma Detect_Blocking is new.

H.6 Pragma Partition_Elaboration_Policy

1/2 {*AI95-00265-01*} This clause defines a pragma for user control over elaboration policy.

Syntax

2/2 {*AI95-00265-01*} The form of a pragma Partition_Elaboration_Policy is as follows:

3/2 **pragma** Partition_Elaboration_Policy (*policy_*identifier);

4/2 The *policy_*identifier shall be either Sequential, Concurrent or an implementation-defined identifier.

4.a/2 **Implementation defined:** Implementation-defined *policy_*identifiers allowed in a pragma Partition_Elaboration_Policy.

Post-Compilation Rules

5/2 {*AI95-00265-01*} {*configuration pragma (Partition_Elaboration_Policy)* [partial]} {*pragma, configuration (Partition_Elaboration_Policy)* [partial]} A pragma Partition_Elaboration_Policy is a configuration pragma. It specifies the elaboration policy for a partition. At most one elaboration policy shall be specified for a partition.

6/2 {*AI95-00265-01*} If the Sequential policy is specified for a partition then pragma Restrictions (No_Task_Hierarchy) shall also be specified for the partition.

Dynamic Semantics

7/2 {*AI95-00265-01*} Notwithstanding what this International Standard says elsewhere, this pragma allows partition elaboration rules concerning task activation and interrupt attachment to be changed. If the *policy_*identifier is Concurrent, or if there is no pragma Partition_Elaboration_Policy defined for the partition, then the rules defined elsewhere in this Standard apply.

8/2 {*AI95-00265-01*} {*AI95-00421-01*} If the partition elaboration policy is Sequential, then task activation and interrupt attachment are performed in the following sequence of steps:

- 9/2 The activation of all library-level tasks and the attachment of interrupt handlers are deferred until all library units are elaborated.
- 10/2 The interrupt handlers are attached by the environment task.
- 11/2 The environment task is suspended while the library-level tasks are activated.
- 12/2 The environment task executes the main subprogram (if any) concurrently with these executing tasks.

13/2 {*AI95-00265-01*} {*AI95-00421-01*} If several dynamic interrupt handler attachments for the same interrupt are deferred, then the most recent call of Attach_Handler or Exchange_Handler determines which handler is attached.

14/2 {*AI95-00265-01*} {*AI95-00421-01*} If any deferred task activation fails, Tasking_Error is raised at the beginning of the sequence of statements of the body of the environment task prior to calling the main subprogram.

Implementation Advice

15/2 {*AI95-00265-01*} If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration then the partition is deadlocked and it is recommended that the partition be immediately terminated.

15.a/2 **Implementation Advice:** If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration then the partition should be immediately terminated.

Implementation Permissions

16/2 {*AI95-00265-01*} If the partition elaboration policy is Sequential and any task activation fails then an implementation may immediately terminate the active partition to mitigate the hazard posed by continuing to execute with a subset of the tasks being active.

NOTES

17/2 12 {*AI95-00421-01*} If any deferred task activation fails, the environment task is unable to handle the Tasking_Error exception and completes immediately. By contrast, if the partition elaboration policy is Concurrent, then this exception could be handled within a library unit.

Extensions to Ada 95

17.a/2 {*AI95-00265-01*} {*AI95-00421-01*} {*extensions to Ada 95*} Pragma Partition_Elaboration_Policy is new.

Annex J
(normative)
Obsolescent Features

{*AI95-00368-01*} [{*obsolescent feature*} This Annex contains descriptions of features of the language whose functionality is largely redundant with other features defined by this International Standard. Use of these features is not recommended in newly written programs. Use of these features can be prevented by using pragma Restrictions (No_Obsolescent_Features), see 13.12.1.] 1/2

Ramification: These features are still part of the language, and have to be implemented by conforming implementations. The primary reason for putting these descriptions here is to get redundant features out of the way of most readers. The designers of the next version of Ada will have to assess whether or not it makes sense to drop these features from the language. 1.a

Wording Changes from Ada 83

The following features have been removed from the language, rather than declared to be obsolescent: 1.b

- The package Low_Level_IO (see A.6). 1.c
- The Epsilon, Mantissa, Emax, Small, Large, Safe_Emax, Safe_Small, and Safe_Large attributes of floating point types (see A.5.3). 1.d
- *This paragraph was deleted.*{*AI95-00284-02*} 1.e/2
- The pragmas System_Name, Storage_Unit, and Memory_Size (see 13.7). 1.f
- The pragma Shared (see C.6). 1.g

Implementations can continue to support the above features for upward compatibility. 1.h

Wording Changes from Ada 95

{*AI95-00368-01*} A mention of the No_Obsolescent_Features restriction was added. 1.i/2

J.1 Renamings of Ada 83 Library Units

Static Semantics

The following library_unit_renaming_declarations exist: 1

2
```
with Ada.Unchecked_Conversion;
generic function Unchecked_Conversion renames Ada.Unchecked_Conversion;
```
3
```
with Ada.Unchecked_Deallocation;
generic procedure Unchecked_Deallocation renames Ada.Unchecked_Deallocation;
```
4
```
with Ada.Sequential_IO;
generic package Sequential_IO renames Ada.Sequential_IO;
```
5
```
with Ada.Direct_IO;
generic package Direct_IO renames Ada.Direct_IO;
```
6
```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```
7
```
with Ada.IO_Exceptions;
package IO_Exceptions renames Ada.IO_Exceptions;
```
8
```
with Ada.Calendar;
package Calendar renames Ada.Calendar;
```
9
```
with System.Machine_Code;
package Machine_Code renames System.Machine_Code; -- If supported.
```

Implementation Requirements

The implementation shall allow the user to replace these renamings. 10

J.2 Allowed Replacements of Characters

Syntax

1 The following replacements are allowed for the vertical line, number sign, and quotation mark characters:

2 • A vertical line character (|) can be replaced by an exclamation mark (!) where used as a delimiter.

3 • The number sign characters (#) of a based_literal can be replaced by colons (:) provided that the replacement is done for both occurrences.

3.a/2 **To be honest:** {*AI95-00285-01*} The intent is that such a replacement works in the Value, Wide_Value, and Wide_Wide_Value attributes, and in the Get procedures of Text_IO (and Wide_Text_IO and Wide_Wide_Text_IO as well)}, so that things like “16:.123:” is acceptable.

4 • The quotation marks (") used as string brackets at both ends of a string literal can be replaced by percent signs (%) provided that the enclosed sequence of characters contains no quotation mark, and provided that both string brackets are replaced. Any percent sign within the sequence of characters shall then be doubled and each such doubled percent sign is interpreted as a single percent sign character value.

5 These replacements do not change the meaning of the program.

5.a **Reason:** The original purpose of this feature was to support hardware (for example, teletype machines) that has long been obsolete. The feature is no longer necessary for that reason. Another use of the feature has been to replace the vertical line character (|) when using certain hardware that treats that character as a (non-English) letter. The feature is no longer necessary for that reason, either, since Ada 95 has full support for international character sets. Therefore, we believe this feature is no longer necessary.

5.b Users of equipment that still uses | to represent a letter will continue to do so. Perhaps by next the time Ada is revised, such equipment will no longer be in use.

5.c Note that it was never legal to use this feature as a convenient method of including double quotes in a string without doubling them — the string literal:

5.d
```
%"This is quoted."%
```

5.e is not legal in Ada 83, nor will it be in Ada 95. One has to write:

5.f
```
"""This is quoted."""
```

J.3 Reduced Accuracy Subtypes

1 A digits_constraint may be used to define a floating point subtype with a new value for its requested decimal precision, as reflected by its Digits attribute. Similarly, a delta_constraint may be used to define an ordinary fixed point subtype with a new value for its *delta*, as reflected by its Delta attribute.

1.a **Discussion:** It might be more direct to make these attributes specifiable via an attribute_definition_clause, and eliminate the syntax for these _constraints.

Syntax

2 delta_constraint ::= **delta** *static*_expression [range_constraint]

Name Resolution Rules

3 {*expected type (delta_constraint expression)* [partial]} The expression of a delta_constraint is expected to be of any real type.

Legality Rules

4 The expression of a delta_constraint shall be static.

For a subtype_indication with a delta_constraint, the subtype_mark shall denote an ordinary fixed point subtype. 5

{*notwithstanding*} For a subtype_indication with a digits_constraint, the subtype_mark shall denote either a decimal fixed point subtype or a floating point subtype (notwithstanding the rule given in 3.5.9 that only allows a decimal fixed point subtype). 6

This paragraph was deleted.{*AI95-00114-01*} 6.a/2

Static Semantics

A subtype_indication with a subtype_mark that denotes an ordinary fixed point subtype and a delta_constraint defines an ordinary fixed point subtype with a *delta* given by the value of the expression of the delta_constraint. If the delta_constraint includes a range_constraint, then the ordinary fixed point subtype is constrained by the range_constraint. 7

A subtype_indication with a subtype_mark that denotes a floating point subtype and a digits_constraint defines a floating point subtype with a requested decimal precision (as reflected by its Digits attribute) given by the value of the expression of the digits_constraint. If the digits_constraint includes a range_-constraint, then the floating point subtype is constrained by the range_constraint. 8

Dynamic Semantics

{*compatibility (delta_constraint with an ordinary fixed point subtype)* [partial]} A delta_constraint is *compatible* with an ordinary fixed point subtype if the value of the expression is no less than the *delta* of the subtype, and the range_constraint, if any, is compatible with the subtype. 9

{*compatibility (digits_constraint with a floating point subtype)* [partial]} A digits_constraint is *compatible* with a floating point subtype if the value of the expression is no greater than the requested decimal precision of the subtype, and the range_constraint, if any, is compatible with the subtype. 10

{*elaboration (delta_constraint)* [partial]} The elaboration of a delta_constraint consists of the elaboration of the range_constraint, if any. 11

Reason: A numeric subtype is considered “constrained” only if a range constraint applies to it. The only effect of a digits_constraint or a delta_constraint without a range_constraint is to specify the value of the corresponding Digits or Delta attribute in the new subtype. The set of values of the subtype is not “constrained” in any way by such _constraints. 11.a

Wording Changes from Ada 83

In Ada 83, a delta_constraint is called a fixed_point_constraint, and a digits_constraint is called a floating_point_constraint. We have adopted other terms because digits_constraints apply primarily to decimal fixed point types now (they apply to floating point types only as an obsolescent feature). 11.b

J.4 The Constrained Attribute

Static Semantics

For every private subtype S, the following attribute is defined: 1

Discussion: This includes generic formal private subtypes. 1.a

S'Constrained 2

Yields the value False if S denotes an unconstrained nonformal private subtype with discriminants; also yields the value False if S denotes a generic formal private subtype, and the associated actual subtype is either an unconstrained subtype with discriminants or an unconstrained array subtype; yields the value True otherwise. The value of this attribute is of the predefined subtype Boolean.

2.a **Reason:** Because Ada 95 has unknown_discriminant_parts, the Constrained attribute of private subtypes is obsolete. This is fortunate, since its Ada 83 definition was confusing, as explained below. Because this attribute is obsolete, we do not bother to extend its definition to private extensions.

2.b The Constrained attribute of an object is *not* obsolete.

2.c Note well: S'Constrained matches the Ada 95 definition of "constrained" only for composite subtypes. For elementary subtypes, S'Constrained is always true, whether or not S is constrained. (The Constrained attribute of an object does not have this problem, as it is only defined for objects of a discriminated type.) So one should think of its designator as being 'Constrained_Or_Elementary.

J.5 ASCII

Static Semantics

1 The following declaration exists in the declaration of package Standard:

```
2   package ASCII is
3     -- Control characters:
4     NUL   : constant Character := nul;    SOH   : constant Character := soh;
      STX   : constant Character := stx;    ETX   : constant Character := etx;
      EOT   : constant Character := eot;    ENQ   : constant Character := enq;
      ACK   : constant Character := ack;    BEL   : constant Character := bel;
      BS    : constant Character := bs;     HT    : constant Character := ht;
      LF    : constant Character := lf;     VT    : constant Character := vt;
      FF    : constant Character := ff;     CR    : constant Character := cr;
      SO    : constant Character := so;     SI    : constant Character := si;
      DLE   : constant Character := dle;    DC1   : constant Character := dc1;
      DC2   : constant Character := dc2;    DC3   : constant Character := dc3;
      DC4   : constant Character := dc4;    NAK   : constant Character := nak;
      SYN   : constant Character := syn;    ETB   : constant Character := etb;
      CAN   : constant Character := can;    EM    : constant Character := em;
      SUB   : constant Character := sub;    ESC   : constant Character := esc;
      FS    : constant Character := fs;     GS    : constant Character := gs;
      RS    : constant Character := rs;     US    : constant Character := us;
      DEL   : constant Character := del;
5     -- Other characters:
6     Exclam   : constant Character:= '!'; Quotation : constant Character:= '"';
      Sharp    : constant Character:= '#'; Dollar    : constant Character:= '$';
      Percent  : constant Character:= '%'; Ampersand : constant Character:= '&';
      Colon    : constant Character:= ':'; Semicolon : constant Character:= ';';
      Query    : constant Character:= '?'; At_Sign   : constant Character:= '@';
      L_Bracket: constant Character:= '['; Back_Slash: constant Character:= '\';
      R_Bracket: constant Character:= ']'; Circumflex: constant Character:= '^';
      Underline: constant Character:= '_'; Grave     : constant Character:= '`';
      L_Brace  : constant Character:= '{'; Bar       : constant Character:= '|';
      R_Brace  : constant Character:= '}'; Tilde     : constant Character:= '~';
7     -- Lower case letters:
8     LC_A: constant Character:= 'a';
      ...
      LC_Z: constant Character:= 'z';
9   end ASCII;
```

J.6 Numeric_Error

Static Semantics

1 The following declaration exists in the declaration of package Standard:

```
2   Numeric_Error : exception renames Constraint_Error;
```

2.a **Discussion:** This is true even though it is not shown in A.1.

Reason: In Ada 83, it was unclear which situations should raise Numeric_Error, and which should raise Constraint_Error. The permissions of RM83-11.6 could often be used to allow the implementation to raise Constraint_Error in a situation where one would normally expect Numeric_Error. To avoid this confusion, all situations that raise Numeric_Error in Ada 83 are changed to raise Constraint_Error in Ada 95. Numeric_Error is changed to be a renaming of Constraint_Error to avoid most of the upward compatibilities associated with this change. 2.b

In new code, Constraint_Error should be used instead of Numeric_Error. 2.c

J.7 At Clauses

Syntax

at_clause ::= **for** direct_name **use at** expression; 1

Static Semantics

An at_clause of the form "for *x* use at *y*;" is equivalent to an attribute_definition_clause of the form "for *x*'Address use *y*;". 2

Reason: The preferred syntax for specifying the address of an entity is an attribute_definition_clause specifying the Address attribute. Therefore, the special-purpose at_clause syntax is now obsolete. 2.a

The above equivalence implies, for example, that only one at_clause is allowed for a given entity. Similarly, it is illegal to give both an at_clause and an attribute_definition_clause specifying the Address attribute. 2.b

Extensions to Ada 83

{*extensions to Ada 83*} We now allow to define the address of an entity using an attribute_definition_clause. This is because Ada 83's at_clause is so hard to remember: programmers often tend to write "for X'Address use...;". 2.c

Wording Changes from Ada 83

Ada 83's address_clause is now called an at_clause to avoid confusion with the new term "Address clause" (that is, an attribute_definition_clause for the Address attribute). 2.d

J.7.1 Interrupt Entries

[Implementations are permitted to allow the attachment of task entries to interrupts via the address clause. Such an entry is referred to as an *interrupt entry*. 1

The address of the task entry corresponds to a hardware interrupt in an implementation-defined manner. (See Ada.Interrupts.Reference in C.3.2.)] 2

Static Semantics

The following attribute is defined: 3

For any task entry X: 4

{*interrupt entry*} X'Address 5

For a task entry whose address is specified (an *interrupt entry*), the value refers to the corresponding hardware interrupt. For such an entry, as for any other task entry, the meaning of this value is implementation defined. The value of this attribute is of the type of the subtype System.Address.

{*specifiable (of Address for entries)* [partial]} Address may be specified for single entries via an attribute_definition_clause. 6

Reason: Because of the equivalence of at_clauses and attribute_definition_clauses, an interrupt entry may be specified via either notation. 6.a

Dynamic Semantics

7 {*initialization (of a task object)* [partial]} As part of the initialization of a task object, the address clause for an interrupt entry is elaborated[, which evaluates the expression of the address clause]. A check is made that the address specified is associated with some interrupt to which a task entry may be attached. {*Program_Error (raised by failure of run-time check)*} If this check fails, Program_Error is raised. Otherwise, the interrupt entry is attached to the interrupt associated with the specified address.

8 {*finalization (of a task object)* [partial]} Upon finalization of the task object, the interrupt entry, if any, is detached from the corresponding interrupt and the default treatment is restored.

9 While an interrupt entry is attached to an interrupt, the interrupt is reserved (see C.3).

10 An interrupt delivered to a task entry acts as a call to the entry issued by a hardware task whose priority is in the System.Interrupt_Priority range. It is implementation defined whether the call is performed as an ordinary entry call, a timed entry call, or a conditional entry call; which kind of call is performed can depend on the specific interrupt.

Bounded (Run-Time) Errors

11 {*bounded error (cause)* [partial]} It is a bounded error to evaluate E'Caller (see C.7.1) in an accept_statement for an interrupt entry. The possible effects are the same as for calling Current_Task from an entry body.

Documentation Requirements

12 The implementation shall document to which interrupts a task entry may be attached.

12.a/2 **Documentation Requirement:** The interrupts to which a task entry may be attached.

13 The implementation shall document whether the invocation of an interrupt entry has the effect of an ordinary entry call, conditional call, or a timed call, and whether the effect varies in the presence of pending interrupts.

13.a/2 **Documentation Requirement:** The type of entry call invoked for an interrupt entry.

Implementation Permissions

14 The support for this subclause is optional.

15 Interrupts to which the implementation allows a task entry to be attached may be designated as reserved for the entire duration of program execution[; that is, not just when they have an interrupt entry attached to them].

16/1 {*8652/0077*} {*AI95-00111-01*} Interrupt entry calls may be implemented by having the hardware execute directly the appropriate accept_statement. Alternatively, the implementation is allowed to provide an internal interrupt handler to simulate the effect of a normal task calling the entry.

17 The implementation is allowed to impose restrictions on the specifications and bodies of tasks that have interrupt entries.

18 It is implementation defined whether direct calls (from the program) to interrupt entries are allowed.

19 If a select_statement contains both a terminate_alternative and an accept_alternative for an interrupt entry, then an implementation is allowed to impose further requirements for the selection of the terminate_alternative in addition to those given in 9.3.

NOTES

20/1 1 {*8652/0077*} {*AI95-00111-01*} Queued interrupts correspond to ordinary entry calls. Interrupts that are lost if not immediately processed correspond to conditional entry calls. It is a consequence of the priority rules that an accept_statement executed in response to an interrupt can be executed with the active priority at which the hardware generates the interrupt, taking precedence over lower priority tasks, without a scheduling action.

2 Control information that is supplied upon an interrupt can be passed to an associated interrupt entry as one or more parameters of mode **in**. 21

Examples

Example of an interrupt entry: 22

23
```
task Interrupt_Handler is
  entry Done;
  for Done'Address use Ada.Interrupts.Reference(Ada.Interrupts.Names.Device_Done);
end Interrupt_Handler;
```

Wording Changes from Ada 83

{*AI95-00114-01*} RM83-13.5.1 did not adequately address the problems associated with interrupts. This feature is now obsolescent and is replaced by the Ada 95 interrupt model as specified in the Systems Programming Annex. 23.a/2

Wording Changes from Ada 95

{*8652/0077*} {*AI95-00111-01*} **Corrigendum:** The undefined term *accept body* was replaced by accept_statement. 23.b/2

J.8 Mod Clauses

Syntax

mod_clause ::= **at mod** *static*_expression; 1

Static Semantics

A record_representation_clause of the form: 2

3
```
for r use
    record at mod a
        ...
    end record;
```

is equivalent to: 4

5
```
for r'Alignment use a;
for r use
    record
        ...
    end record;
```

Reason: The preferred syntax for specifying the alignment of an entity is an attribute_definition_clause specifying the Alignment attribute. Therefore, the special-purpose mod_clause syntax is now obsolete. 5.a

The above equivalence implies, for example, that it is illegal to give both a mod_clause and an attribute_definition_clause specifying the Alignment attribute for the same type. 5.b

Wording Changes from Ada 83

Ada 83's alignment_clause is now called a mod_clause to avoid confusion with the new term “Alignment clause” (that is, an attribute_definition_clause for the Alignment attribute). 5.c

J.9 The Storage_Size Attribute

Static Semantics

For any task subtype T, the following attribute is defined: 1

T'Storage_Size 2

Denotes an implementation-defined value of type *universal_integer* representing the number of storage elements reserved for a task of the subtype T.

2.a **To be honest:** T'Storage_Size cannot be particularly meaningful in the presence of a pragma Storage_Size, especially when the expression is dynamic, or depends on a discriminant of the task, because the Storage_Size will be different for different objects of the type. Even without such a pragma, the Storage_Size can be different for different objects of the type, and in any case, the value is implementation defined. Hence, it is always implementation defined.

3/2 {*AI95-00345-01*} {*specifiable (of Storage_Size for a task first subtype)* [partial]} Storage_Size may be specified for a task first subtype that is not an interface via an attribute_definition_clause.

Wording Changes from Ada 95

3.a/2 {*AI95-00345-01*} We don't allow specifying Storage_Size on task interfaces. We don't need to mention class-wide task types, because these cannot be a first subtype.

J.10 Specific Suppression of Checks

1/2 {*AI95-00224-01*} Pragma Suppress can be used to suppress checks on specific entities.

Syntax

2/2 {*AI95-00224-01*} The form of a specific Suppress pragma is as follows:

3/2 **pragma** Suppress(identifier, [On =>] name);

Legality Rules

4/2 {*AI95-00224-01*} The identifier shall be the name of a check (see 11.5). The name shall statically denote some entity.

5/2 {*AI95-00224-01*} For a specific Suppress pragma that is immediately within a package_specification, the name shall denote an entity (or several overloaded subprograms) declared immediately within the package_specification.

Static Semantics

6/2 {*AI95-00224-01*} A specific Suppress pragma applies to the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a package_specification, to the end of the scope of the named entity. The pragma applies only to the named entity, or, for a subtype, on objects and values of its type. A specific Suppress pragma suppresses the named check for any entities to which it applies (see 11.5). Which checks are associated with a specific entity is not defined by this International Standard.

6.a/2 **Discussion:** The language doesn't specify exactly which entities control whether a check is performed. For example, in

6.b
```
pragma Suppress (Range_Check, On => A);
A := B;
```

6.c whether or not the range check is performed is not specified. The compiler may require that checks are suppressed on B or on the type of A in order to omit the range check.

Implementation Permissions

7/2 {*AI95-00224-01*} An implementation is allowed to place restrictions on specific Suppress pragmas.

NOTES

8/2 3 {*AI95-00224-01*} An implementation may support a similar On parameter on pragma Unsuppress (see 11.5).

Wording Changes from Ada 95

8.a/2 {*AI95-00224-01*} This clause is new. This feature was moved here because it is important for pragma Unsuppress that there be an unambiguous meaning for each checking pragma. For instance, in the example

8.b
```
pragma Suppress (Range_Check);
pragma Unsuppress (Range_Check, On => A);
A := B;
```

the user needs to be able to depend on the range check being made on the assignment. But a compiler survey showed that the interpretation of this feature varied widely; trying to define this carefully was likely to cause a lot of user and implementer pain. Thus the feature was moved here, to emphasize that its use is not portable. 8.c

J.11 The Class Attribute of Untagged Incomplete Types

Static Semantics

{*AI95-00326-01*} For the first subtype S of a type *T* declared by an incomplete_type_declaration that is not tagged, the following attribute is defined: 1/2

{*AI95-00326-01*} S'Class 2/2

Denotes the first subtype of the incomplete class-wide type rooted at *T*. The completion of *T* shall declare a tagged type. Such an attribute reference shall occur in the same library unit as the incomplete_type_declaration.

Reason: {*AI95-00326-01*} This must occur in the same unit to prevent children from imposing requirements on their ancestor library units for deferred incomplete types. 2.a/2

Wording Changes from Ada 95

{*AI95-00326-01*} This clause is new. This feature was moved here because the tagged incomplete type provides a better way to provide this capability (it doesn't put requirements on the completion based on uses that could be anywhere). Pity we didn't think of it in 1994. 2.b/2

J.12 Pragma Interface

Syntax

{*AI95-00284-02*} In addition to an identifier, the reserved word **interface** is allowed as a pragma name, to provide compatibility with a prior edition of this International Standard. 1/2

Implementation Note: {*AI95-00284-02*} All implementations need to at least recognize and ignore this pragma. A syntax error is not an acceptable implementation of this pragma. 1.a/2

Wording Changes from Ada 95

{*AI95-00326-01*} This clause is new. This is necessary as **interface** is now a reserved word, which would prevent pragma Interface from being an implementation-defined pragma. We don't define any semantics for this pragma, as we expect that implementations will continue to use whatever they currently implement - requiring any changes would be counter-productive. 1.b/2

J.13 Dependence Restriction Identifiers

{*AI95-00394-01*} The following restrictions involve dependence on specific language-defined units. The more general restriction No_Dependence (see 13.12.1) should be used for this purpose. 1/2

Static Semantics

{*AI95-00394-01*} The following *restriction_*identifiers exist: 2/2

{*AI95-00394-01*} {*Restrictions (No_Asynchronous_Control)*} No_Asynchronous_Control 3/2

Semantic dependence on the predefined package Asynchronous_Task_Control is not allowed.

{*AI95-00394-01*} {*Restrictions (No_Unchecked_Conversion)*} No_Unchecked_Conversion 4/2

Semantic dependence on the predefined generic function Unchecked_Conversion is not allowed.

{*AI95-00394-01*} {*Restrictions (No_Unchecked_Deallocation)*} No_Unchecked_Deallocation 5/2

Semantic dependence on the predefined generic procedure Unchecked_Deallocation is not allowed.

Wording Changes from Ada 95

5.a/2 {*AI95-00394-01*} This clause is new. These restrictions are replaced by the more general No_Dependence (see 13.12.1).

J.14 Character and Wide_Character Conversion Functions

Static Semantics

1/2 {*AI95-00395-01*} The following declarations exist in the declaration of package Ada.Characters.Handling:

2/2
```
function Is_Character (Item : in Wide_Character) return Boolean
   renames Conversions.Is_Character;
function Is_String    (Item : in Wide_String)    return Boolean
   renames Conversions.Is_String;
```

3/2
```
function To_Character (Item       : in Wide_Character;
                       Substitute : in Character := ' ')
                       return Character
   renames Conversions.To_Character;
```

4/2
```
function To_String    (Item       : in Wide_String;
                        Substitute : in Character := ' ')
                        return String
   renames Conversions.To_String;
```

5/2
```
function To_Wide_Character (Item : in Character) return Wide_Character
   renames Conversions.To_Wide_Character;
```

6/2
```
function To_Wide_String    (Item : in String)    return Wide_String
   renames Conversions.To_Wide_String;
```

Wording Changes from Ada 95

6.a/2 {*AI95-00394-01*} This clause is new. These subprograms were moved to Characters.Conversions (see A.3.4).

Annex K (informative) Language-Defined Attributes

{*attribute*} This annex summarizes the definitions given elsewhere of the language-defined attributes. 1

P'Access — For a prefix P that denotes a subprogram: 2

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. See 3.10.2. 3

X'Access — For a prefix X that denotes an aliased view of an object: 4

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. See 3.10.2. 5

X'Address — For a prefix X that denotes an object, program unit, or label: 6/1

Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address. See 13.3. 7

S'Adjacent — For every subtype S of a floating point type *T*: 8

S'Adjacent denotes a function with the following specification: 9

```
function S'Adjacent (X, Towards : T)
  return T
```
10

{*Constraint_Error (raised by failure of run-time check)*} If *Towards* = *X*, the function yields *X*; otherwise, it yields the machine number of the type *T* adjacent to *X* in the direction of *Towards*, if that machine number exists. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} If the result would be outside the base range of S, Constraint_Error is raised. When *T*'Signed_Zeros is True, a zero result has the sign of *X*. When *Towards* is zero, its sign has no bearing on the result. See A.5.3. 11

S'Aft — For every fixed point subtype S: 12

S'Aft yields the number of decimal digits needed after the decimal point to accommodate the *delta* of the subtype S, unless the *delta* of the subtype S is greater than 0.1, in which case the attribute yields the value one. (S'Aft is the smallest positive integer N for which (10**N)*S'Delta is greater than or equal to one.) The value of this attribute is of the type *universal_integer*. See 3.5.10. 13

S'Alignment 13.1/2
For every subtype S:

The value of this attribute is of type *universal_integer*, and nonnegative. 13.2/2

For an object X of subtype S, if S'Alignment is not zero, then X'Alignment is a nonzero integral multiple of S'Alignment unless specified otherwise by a representation item. See 13.3. 13.3/2

X'Alignment 14/1
For a prefix X that denotes an object:

The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If X'Alignment is not zero, then X is aligned on a storage unit boundary and X'Address is an integral multiple of X'Alignment (that is, the Address modulo the Alignment is zero). 15

16/2 See 13.3.

17 S'Base For every scalar subtype S:

18 S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the *base subtype* of the type. See 3.5.

19 S'Bit_Order
For every specific record subtype S:

20 Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. See 13.5.3.

21/1 P'Body_Version
For a prefix P that statically denotes a program unit:

22 Yields a value of the predefined type String that identifies the version of the compilation unit that contains the body (but not any subunits) of the program unit. See E.3.

23 T'Callable For a prefix T that is of a task type (after any implicit dereference):

24 Yields the value True when the task denoted by T is *callable*, and False otherwise; See 9.9.

25 E'Caller For a prefix E that denotes an entry_declaration:

26 Yields a value of the type Task_Id that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an entry_body or accept_statement corresponding to the entry_declaration denoted by E. See C.7.1.

27 S'Ceiling For every subtype S of a floating point type *T*:

28 S'Ceiling denotes a function with the following specification:

29
```
function S'Ceiling (X : T)
  return T
```

30 The function yields the value $\lceil X \rceil$, i.e., the smallest (most negative) integral value greater than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a negative sign when S'Signed_Zeros is True. See A.5.3.

31 S'Class For every subtype S of an untagged private type whose full view is tagged:

32 Denotes the class-wide subtype corresponding to the full view of S. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. After the full view, the Class attribute of the full view can be used. See 7.3.1.

33 S'Class For every subtype S of a tagged type *T* (specific or class-wide):

34 S'Class denotes a subtype of the class-wide type (called *T*'Class in this International Standard) for the class rooted at *T* (or if S already denotes a class-wide subtype, then S'Class is the same as S).

35 {*unconstrained (subtype)*} {*constrained (subtype)*} S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type *T* belong to S. See 3.9.

36/1 X'Component_Size
For a prefix X that denotes an array subtype or array object (after any implicit dereference):

37 Denotes the size in bits of components of the type of X. The value of this attribute is of type *universal_integer*. See 13.3.

38 S'Compose For every subtype S of a floating point type *T*:

39 S'Compose denotes a function with the following specification:

```
function S'Compose (Fraction : T;
                    Exponent : universal_integer)
  return T
```
40

{*Constraint_Error (raised by failure of run-time check)*} Let *v* be the value *Fraction* · *T*'Machine_Radix$^{Exponent-k}$, where *k* is the normalized exponent of *Fraction*. If *v* is a machine number of the type *T*, or if |*v*| ≥ *T*'Model_Small, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *Fraction* when S'Signed_Zeros is True. See A.5.3. 41

A'Constrained 42

For a prefix A that is of a discriminated type (after any implicit dereference):

Yields the value True if A denotes a constant, a value, or a constrained variable, and False otherwise. See 3.7.2. 43

S'Copy_Sign 44

For every subtype S of a floating point type *T*:

S'Copy_Sign denotes a function with the following specification: 45

```
function S'Copy_Sign (Value, Sign : T)
  return T
```
46

{*Constraint_Error (raised by failure of run-time check)*} If the value of *Value* is nonzero, the function yields a result whose magnitude is that of *Value* and whose sign is that of *Sign*; otherwise, it yields the value zero. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is optionally raised if the result is outside the base range of S. A zero result has the sign of *Sign* when S'Signed_Zeros is True. See A.5.3. 47

E'Count — For a prefix E that denotes an entry of a task or protected unit: 48

Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type *universal_integer*. See 9.9. 49

S'Definite — For a prefix S that denotes a formal indefinite subtype: 50/1

S'Definite yields True if the actual subtype corresponding to S is definite; otherwise it yields False. The value of this attribute is of the predefined type Boolean. See 12.5.1. 51

S'Delta — For every fixed point subtype S: 52

S'Delta denotes the *delta* of the fixed point subtype S. The value of this attribute is of the type *universal_real*. See 3.5.10. 53

S'Denorm — For every subtype S of a floating point type *T*: 54

Yields the value True if every value expressible in the form 55

± *mantissa* · *T*'Machine_Radix$^{T\text{'Machine\_Emin}}$

where *mantissa* is a nonzero *T*'Machine_Mantissa-digit fraction in the number base *T*'Machine_Radix, the first digit of which is zero, is a machine number (see 3.5.7) of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

S'Digits — For every decimal fixed point subtype S: 56

S'Digits denotes the *digits* of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type *universal_integer*. See 3.5.10. 57

S'Digits — For every floating point subtype S: 58

S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. See 3.5.8. 59

60 S'Exponent For every subtype S of a floating point type *T*:

61 S'Exponent denotes a function with the following specification:

62
```
function S'Exponent (X : T)
  return universal_integer
```

63 The function yields the normalized exponent of *X*. See A.5.3.

64 S'External_Tag
For every subtype S of a tagged type *T* (specific or class-wide):

65 {*External_Tag clause*} {*specifiable (of External_Tag for a tagged type)* [partial]} S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined. See 3.9.2 and 13.13.2. See 13.3.

66/1 A'First For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

67 A'First denotes the lower bound of the first index range; its type is the corresponding index type. See 3.6.2.

68 S'First For every scalar subtype S:

69 S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S. See 3.5.

70/1 A'First(N)
For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

71 A'First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type. See 3.6.2.

72 R.C'First_Bit
For a component C of a composite, non-array object R:

73/2 If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the first_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*. See 13.5.2.

74 S'Floor For every subtype S of a floating point type *T*:

75 S'Floor denotes a function with the following specification:

76
```
function S'Floor (X : T)
  return T
```

77 The function yields the value $\lfloor X \rfloor$, i.e., the largest (most positive) integral value less than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a positive sign. See A.5.3.

78 S'Fore For every fixed point subtype S:

79 S'Fore yields the minimum number of characters needed before the decimal point for the decimal representation of any value of the subtype S, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute is of the type *universal_integer*. See 3.5.10.

80 S'Fraction For every subtype S of a floating point type *T*:

S'Fraction denotes a function with the following specification: 81

```
function S'Fraction (X : T)
  return T
```
82

The function yields the value $X \cdot T$'Machine_Radix^{-k}, where k is the normalized exponent of X. A zero result, which can only occur when X is zero, has the sign of X. See A.5.3. 83

T'Identity — For a prefix T that is of a task type (after any implicit dereference): 84

Yields a value of the type Task_Id that identifies the task denoted by T. See C.7.1. 85

E'Identity — For a prefix E that denotes an exception: 86/1

E'Identity returns the unique identity of the exception. The type of this attribute is Exception_Id. See 11.4.1. 87

S'Image — For every scalar subtype S: 88

S'Image denotes a function with the following specification: 89

```
function S'Image(Arg : S'Base)
  return String
```
90

The function returns an image of the value of *Arg* as a String. See 3.5. 91/2

S'Class'Input — 92
For every subtype S'Class of a class-wide type *T*'Class:

S'Class'Input denotes a function with the following specification: 93

```
function S'Class'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T'Class
```
94/2

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which might raise Tag_Error — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. If the specific type identified by the internal tag is not covered by *T*'Class or is abstract, Constraint_Error is raised. See 13.13.2. 95/2

S'Input — For every subtype S of a specific type *T*: 96

S'Input denotes a function with the following specification: 97

```
function S'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T
```
98/2

S'Input reads and returns one value from *Stream*, using any bounds or discriminants written by a corresponding S'Output to determine how much to read. See 13.13.2. 99

A'Last — For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 100/1

A'Last denotes the upper bound of the first index range; its type is the corresponding index type. See 3.6.2. 101

S'Last — For every scalar subtype S: 102

S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S. See 3.5. 103

A'Last(N) — For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 104/1

A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type. See 3.6.2. 105

106 R.C'Last_Bit

For a component C of a composite, non-array object R:

107/2 If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the last_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer*. See 13.5.2.

108 S'Leading_Part

For every subtype S of a floating point type *T*:

109 S'Leading_Part denotes a function with the following specification:

110
```
function S'Leading_Part (X : T;
                         Radix_Digits : universal_integer)
  return T
```

111 Let *v* be the value *T*'Machine_Radix$^{k-Radix\_Digits}$, where *k* is the normalized exponent of *X*. The function yields the value

112 • $\lfloor X/v \rfloor \cdot v$, when *X* is nonnegative and *Radix_Digits* is positive;

113 • $\lceil X/v \rceil \cdot v$, when *X* is negative and *Radix_Digits* is positive.

114 {*Constraint_Error (raised by failure of run-time check)*} {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised when *Radix_Digits* is zero or negative. A zero result, which can only occur when *X* is zero, has the sign of *X*. See A.5.3.

115/1 A'Length For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

116 A'Length denotes the number of values of the first index range (zero for a null range); its type is *universal_integer*. See 3.6.2.

117/1 A'Length(N)

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

118 A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is *universal_integer*. See 3.6.2.

119 S'Machine For every subtype S of a floating point type *T*:

120 S'Machine denotes a function with the following specification:

121
```
function S'Machine (X : T)
  return T
```

122 {*Constraint_Error (raised by failure of run-time check)*} If *X* is a machine number of the type *T*, the function yields *X*; otherwise, it yields the value obtained by rounding or truncating *X* to either one of the adjacent machine numbers of the type *T*. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if rounding or truncating *X* to the precision of the machine numbers results in a value outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

123 S'Machine_Emax

For every subtype S of a floating point type *T*:

124 Yields the largest (most positive) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

S'Machine_Emin 125
For every subtype S of a floating point type *T*:

126 Yields the smallest (most negative) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

S'Machine_Mantissa 127
For every subtype S of a floating point type *T*:

128 Yields the largest value of *p* such that every value expressible in the canonical form (for the type *T*), having a *p*-digit *mantissa* and an *exponent* between *T*'Machine_Emin and *T*'Machine_Emax, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

S'Machine_Overflows 129
For every subtype S of a fixed point type *T*:

130 Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.4.

S'Machine_Overflows 131
For every subtype S of a floating point type *T*:

132 Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

S'Machine_Radix 133
For every subtype S of a fixed point type *T*:

134 Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*. See A.5.4.

S'Machine_Radix 135
For every subtype S of a floating point type *T*:

136 Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*. See A.5.3.

S'Machine_Rounding 136.1/2
For every subtype S of a floating point type *T*:

136.2/2 S'Machine_Rounding denotes a function with the following specification:

136.3/2
```
function S'Machine_Rounding (X : T)
  return T
```

136.4/2 The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of *X* when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor.{*unspecified* [partial]} See A.5.3.

S'Machine_Rounds 137
For every subtype S of a fixed point type *T*:

138 Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.4.

139 S'Machine_Rounds
For every subtype S of a floating point type *T*:

140 Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

141 S'Max For every scalar subtype S:

142 S'Max denotes a function with the following specification:

143
```
function S'Max(Left, Right : S'Base)
  return S'Base
```

144 The function returns the greater of the values of the two parameters. See 3.5.

145 S'Max_Size_In_Storage_Elements
For every subtype S:

146/2 Denotes the maximum value for Size_In_Storage_Elements that could be requested by the implementation via Allocate for an access type whose designated subtype is S. For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then this accounts for any calls on Allocate that could be performed to provide space for such coextensions. The value of this attribute is of type *universal_integer*. See 13.11.1.

147 S'Min For every scalar subtype S:

148 S'Min denotes a function with the following specification:

149
```
function S'Min(Left, Right : S'Base)
  return S'Base
```

150 The function returns the lesser of the values of the two parameters. See 3.5.

150.1/2 S'Mod For every modular subtype S:

150.2/2 S'Mod denotes a function with the following specification:

150.3/2
```
function S'Mod (Arg : universal_integer)
  return S'Base
```

150.4/2 This function returns *Arg* **mod** S'Modulus, as a value of the type of S. See 3.5.4.

151 S'Model For every subtype S of a floating point type *T*:

152 S'Model denotes a function with the following specification:

153
```
function S'Model (X : T)
  return T
```

154 If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. See A.5.3.

155 S'Model_Emin
For every subtype S of a floating point type *T*:

156 If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to the value of *T*'Machine_Emin. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*. See A.5.3.

S'Model_Epsilon 157
For every subtype S of a floating point type *T*:

158 Yields the value *T*'Machine_Radix$^{1 - T\text{'Model\_Mantissa}}$. The value of this attribute is of the type *universal_real*. See A.5.3.

S'Model_Mantissa 159
For every subtype S of a floating point type *T*:

160 If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to $\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix})\rceil + 1$, where *d* is the requested decimal precision of *T*, and less than or equal to the value of *T*'Machine_Mantissa. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*. See A.5.3.

S'Model_Small 161
For every subtype S of a floating point type *T*:

162 Yields the value *T*'Machine_Radix$^{T\text{'Model\_Emin} - 1}$. The value of this attribute is of the type *universal_real*. See A.5.3.

S'Modulus For every modular subtype S: 163

164 S'Modulus yields the modulus of the type of S, as a value of the type *universal_integer*. See 3.5.4.

S'Class'Output 165
For every subtype S'Class of a class-wide type *T*'Class:

166 S'Class'Output denotes a procedure with the following specification:

167/2
```
procedure S'Class'Output(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

168/2 First writes the external tag of *Item* to *Stream* (by calling String'Output(*Stream*, Tags.-External_Tag(*Item*'Tag)) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag. Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S. See 13.13.2.

S'Output For every subtype S of a specific type *T*: 169

170 S'Output denotes a procedure with the following specification:

171/2
```
procedure S'Output(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```

172 S'Output writes the value of *Item* to *Stream*, including any bounds or discriminants. See 13.13.2.

D'Partition_Id 173/1
For a prefix D that denotes a library-level declaration, excepting a declaration of or within a declared-pure library unit:

174 Denotes a value of the type *universal_integer* that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see E.2.3) the given partition is the one where the body of D was elaborated. See E.1.

S'Pos For every discrete subtype S: 175

176 S'Pos denotes a function with the following specification:

177
```
function S'Pos(Arg : S'Base)
  return universal_integer
```

178 This function returns the position number of the value of *Arg*, as a value of type *universal_integer*. See 3.5.5.

179 R.C'Position

For a component C of a composite, non-array object R:

180/2 If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the position of the component_clause; otherwise, denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*. See 13.5.2.

181 S'Pred For every scalar subtype S:

182 S'Pred denotes a function with the following specification:

183
```
function S'Pred(Arg : S'Base)
  return S'Base
```

184 {*Constraint_Error (raised by failure of run-time check)*} For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of *Arg*. For a fixed point type, the function returns the result of subtracting *small* from the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of *Arg*; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if there is no such machine number. See 3.5.

184.1/2 P'Priority For a prefix P that denotes a protected object:

184.2/2 Denotes a non-aliased component of the protected object P. This component is of type System.Any_Priority and its value is the priority of P. P'Priority denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P. See D.5.2.

185/1 A'Range For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

186 A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once. See 3.6.2.

187 S'Range For every scalar subtype S:

188 S'Range is equivalent to the range S'First .. S'Last. See 3.5.

189/1 A'Range(N)

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

190 A'Range(N) is equivalent to the range A'First(N) .. A'Last(N), except that the prefix A is only evaluated once. See 3.6.2.

191 S'Class'Read

For every subtype S'Class of a class-wide type *T*'Class:

192 S'Class'Read denotes a procedure with the following specification:

193/2
```
procedure S'Class'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T'Class)
```

194 Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item. See 13.13.2.

195 S'Read For every subtype S of a specific type *T*:

S'Read denotes a procedure with the following specification: 196

```
procedure S'Read(                                                197/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T)
```

S'Read reads the value of *Item* from *Stream*. See 13.13.2. 198

S'Remainder 199

For every subtype S of a floating point type *T*:

S'Remainder denotes a function with the following specification: 200

```
function S'Remainder (X, Y : T)                                  201
  return T
```

202 {*Constraint_Error (raised by failure of run-time check)*} For nonzero *Y*, let *v* be the value *X* – *n* · *Y*, where *n* is the integer nearest to the exact value of *X*/*Y*; if |*n* – *X*/*Y*| = 1/2, then *n* is chosen to be even. If *v* is a machine number of the type *T*, the function yields *v*; otherwise, it yields zero. {*Division_Check* [partial]} {*check, language-defined (Division_Check)*} Constraint_Error is raised if *Y* is zero. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

S'Round For every decimal fixed point subtype S: 203

S'Round denotes a function with the following specification: 204

```
function S'Round(X : universal_real)                             205
  return S'Base
```

206 The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S). See 3.5.10.

S'Rounding For every subtype S of a floating point type *T*: 207

S'Rounding denotes a function with the following specification: 208

```
function S'Rounding (X : T)                                      209
  return T
```

210 The function yields the integral value nearest to *X*, rounding away from zero if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

S'Safe_First 211

For every subtype S of a floating point type *T*:

212 Yields the lower bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*. See A.5.3.

S'Safe_Last 213

For every subtype S of a floating point type *T*:

214 Yields the upper bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*. See A.5.3.

S'Scale For every decimal fixed point subtype S: 215

216 S'Scale denotes the *scale* of the subtype S, defined as the value N such that S'Delta = 10.0**(–N). {*scale (of a decimal fixed point subtype)*} The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S. The value of this attribute is of the type *universal_integer*. See 3.5.10.

S'Scaling For every subtype S of a floating point type *T*: 217

218 S'Scaling denotes a function with the following specification:

219
```
function S'Scaling (X : T;
                    Adjustment : universal_integer)
  return T
```

220 {*Constraint_Error (raised by failure of run-time check)*} Let *v* be the value $X \cdot T\text{'Machine\_Radix}^{Adjustment}$. If *v* is a machine number of the type *T*, or if $|v| \geq T$'Model_Small, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

221 S'Signed_Zeros
For every subtype S of a floating point type *T*:

222 Yields the value True if the hardware representation for the type *T* has the capability of representing both positively and negatively signed zeros, these being generated and used by the predefined operations of the type *T* as specified in IEC 559:1989; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

223 S'Size For every subtype S:

224 If S is definite, denotes the size (in bits) that the implementation would choose for the following objects of subtype S:

225
- A record component of subtype S when the record type is packed.

226
- The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype.

227 If S is indefinite, the meaning is implementation defined. The value of this attribute is of the type *universal_integer*. See 13.3.

228/1 X'Size For a prefix X that denotes an object:

229 Denotes the size in bits of the representation of the object. The value of this attribute is of the type *universal_integer*. See 13.3.

230 S'Small For every fixed point subtype S:

231 S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. See 3.5.10.

232 S'Storage_Pool
For every access-to-object subtype S:

233 Denotes the storage pool of the type of S. The type of this attribute is Root_Storage_-Pool'Class. See 13.11.

234 S'Storage_Size
For every access-to-object subtype S:

235 Yields the result of calling Storage_Size(S'Storage_Pool), which is intended to be a measure of the number of storage elements reserved for the pool. The type of this attribute is *universal_integer*. See 13.11.

236/1 T'Storage_Size
For a prefix T that denotes a task object (after any implicit dereference):

237 Denotes the number of storage elements reserved for the task. The value of this attribute is of the type *universal_integer*. The Storage_Size includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the “task control block” used by some implementations.) See 13.3.

S'Stream_Size 237.1/2
For every subtype S of an elementary type *T*:

237.2/2 Denotes the number of bits occupied in a stream by items of subtype S. Hence, the number of stream elements required per item of elementary type *T* is:

237.3/2
```
T'Stream_Size / Ada.Streams.Stream_Element'Size
```

237.4/2 The value of this attribute is of type *universal_integer* and is a multiple of Stream_Element'Size. See 13.13.2.

S'Succ 238 For every scalar subtype S:

239 S'Succ denotes a function with the following specification:

240
```
function S'Succ(Arg : S'Base)
  return S'Base
```

241 {*Constraint_Error (raised by failure of run-time check)*} For an enumeration type, the function returns the value whose position number is one more than that of the value of *Arg*; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of adding one to the value of *Arg*. For a fixed point type, the function returns the result of adding *small* to the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately above the value of *Arg*; {*Range_Check* [partial]} {*check, language-defined (Range_Check)*} Constraint_Error is raised if there is no such machine number. See 3.5.

X'Tag 242 For a prefix X that is of a class-wide tagged type (after any implicit dereference):

243 X'Tag denotes the tag of X. The value of this attribute is of type Tag. See 3.9.

S'Tag 244 For every subtype S of a tagged type *T* (specific or class-wide):

245 S'Tag denotes the tag of the type *T* (or if *T* is class-wide, the tag of the root type of the corresponding class). The value of this attribute is of type Tag. See 3.9.

T'Terminated 246
For a prefix T that is of a task type (after any implicit dereference):

247 Yields the value True if the task denoted by T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean. See 9.9.

S'Truncation 248
For every subtype S of a floating point type *T*:

249 S'Truncation denotes a function with the following specification:

250
```
function S'Truncation (X : T)
  return T
```

251 The function yields the value $\lceil X \rceil$ when *X* is negative, and $\lfloor X \rfloor$ otherwise. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

S'Unbiased_Rounding 252
For every subtype S of a floating point type *T*:

253 S'Unbiased_Rounding denotes a function with the following specification:

254
```
function S'Unbiased_Rounding (X : T)
  return T
```

255 The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

X'Unchecked_Access 256
For a prefix X that denotes an aliased view of an object:

257 All rules and semantics that apply to X'Access (see 3.10.2) apply also to X'Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package. See 13.10.

258 S'Val For every discrete subtype S:

259 S'Val denotes a function with the following specification:

260
```
function S'Val(Arg : universal_integer)
  return S'Base
```

261 {*evaluation (Val)* [partial]} {*Constraint_Error (raised by failure of run-time check)*} This function returns a value of the type of S whose position number equals the value of *Arg*. See 3.5.5.

262 X'Valid For a prefix X that denotes a scalar object (after any implicit dereference):

263 Yields True if and only if the object denoted by X is normal and has a valid representation. The value of this attribute is of the predefined type Boolean. See 13.9.2.

264 S'Value For every scalar subtype S:

265 S'Value denotes a function with the following specification:

266
```
function S'Value(Arg : String)
  return S'Base
```

267 This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces. See 3.5.

268/1 P'Version For a prefix P that statically denotes a program unit:

269 Yields a value of the predefined type String that identifies the version of the compilation unit that contains the declaration of the program unit. See E.3.

270 S'Wide_Image
For every scalar subtype S:

271 S'Wide_Image denotes a function with the following specification:

272
```
function S'Wide_Image(Arg : S'Base)
  return Wide_String
```

273/2 {*image (of a value)*} The function returns an image of the value of *Arg* as a Wide_String. See 3.5.

274 S'Wide_Value
For every scalar subtype S:

275 S'Wide_Value denotes a function with the following specification:

276
```
function S'Wide_Value(Arg : Wide_String)
  return S'Base
```

277 This function returns a value given an image of the value as a Wide_String, ignoring any leading or trailing spaces. See 3.5.

277.1/2 S'Wide_Wide_Image
For every scalar subtype S:

277.2/2 S'Wide_Wide_Image denotes a function with the following specification:

277.3/2
```
function S'Wide_Wide_Image(Arg : S'Base)
  return Wide_Wide_String
```

277.4/2 {*image (of a value)*} The function returns an *image* of the value of *Arg*, that is, a sequence of characters representing the value in display form. See 3.5.

277.5/2 S'Wide_Wide_Value
For every scalar subtype S:

277.6/2 S'Wide_Wide_Value denotes a function with the following specification:

```
function S'Wide_Wide_Value(Arg : Wide_Wide_String)
  return S'Base
```
277.7/2

This function returns a value given an image of the value as a Wide_Wide_String, ignoring any leading or trailing spaces. See 3.5. 277.8/2

S'Wide_Wide_Width 277.9/2
For every scalar subtype S:

S'Wide_Wide_Width denotes the maximum length of a Wide_Wide_String returned by S'Wide_Wide_Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5. 277.10/2

S'Wide_Width 278
For every scalar subtype S:

S'Wide_Width denotes the maximum length of a Wide_String returned by S'Wide_Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5. 279

S'Width For every scalar subtype S: 280

S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5. 281

S'Class'Write 282
For every subtype S'Class of a class-wide type *T*'Class:

S'Class'Write denotes a procedure with the following specification: 283

```
procedure S'Class'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```
284/2

Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item. See 13.13.2. 285

S'Write For every subtype S of a specific type *T*: 286

S'Write denotes a procedure with the following specification: 287

```
procedure S'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```
288/2

S'Write writes the value of *Item* to *Stream*. See 13.13.2. 289

Annex L
(informative)
Language-Defined Pragmas

{*pragma*} This Annex summarizes the definitions given elsewhere of the language-defined pragmas. 1

pragma All_Calls_Remote[(*library_unit*_name)]; — See E.2.3. 2

pragma Assert([Check =>] *boolean*_expression[, [Message =>] *string*_expression]); — See 11.4.2. 2.1/2

pragma Assertion_Policy(*policy*_identifier); — See 11.4.2. 2.2/2

pragma Asynchronous(local_name); — See E.4.1. 3

pragma Atomic(local_name); — See C.6. 4

pragma Atomic_Components(*array*_local_name); — See C.6. 5

pragma Attach_Handler(*handler*_name, expression); — See C.3.1. 6

pragma Controlled(*first_subtype*_local_name); — See 13.11.3. 7

pragma Convention([Convention =>] *convention*_identifier,[Entity =>] local_name); — See B.1. 8

pragma Detect_Blocking; — See H.5. 8.1/2

pragma Discard_Names[([On =>] local_name)]; — See C.5. 9

pragma Elaborate(*library_unit*_name{, *library_unit*_name}); — See 10.2.1. 10

pragma Elaborate_All(*library_unit*_name{, *library_unit*_name}); — See 10.2.1. 11

pragma Elaborate_Body[(*library_unit*_name)]; — See 10.2.1. 12

pragma Export(13
 [Convention =>] *convention*_identifier, [Entity =>] local_name
 [, [External_Name =>] *string*_expression] [, [Link_Name =>] *string*_expression]); — See B.1.

pragma Import(14
 [Convention =>] *convention*_identifier, [Entity =>] local_name
 [, [External_Name =>] *string*_expression] [, [Link_Name =>] *string*_expression]); — See B.1.

pragma Inline(name {, name}); — See 6.3.2. 15

pragma Inspection_Point[(*object*_name {, *object*_name})]; — See H.3.2. 16

pragma Interrupt_Handler(*handler*_name); — See C.3.1. 17

pragma Interrupt_Priority[(expression)]; — See D.1. 18

pragma Linker_Options(*string*_expression); — See B.1. 19

pragma List(identifier); — See 2.8. 20

pragma Locking_Policy(*policy*_identifier); — See D.3. 21

pragma No_Return(*procedure*_local_name{, *procedure*_local_name}); — See 6.5.1. 21.1/2

pragma Normalize_Scalars; — See H.1. 22

pragma Optimize(identifier); — See 2.8. 23

24 **pragma** Pack(*first_subtype*_local_name); — See 13.2.

25 **pragma** Page; — See 2.8.

25.1/2 **pragma** Partition_Elaboration_Policy (*policy*_identifier); — See H.6.

25.2/2 **pragma** Preelaborable_Initialization(direct_name); — See 10.2.1.

26 **pragma** Preelaborate[(*library_unit*_name)]; — See 10.2.1.

27 **pragma** Priority(expression); — See D.1.

27.1/2 **pragma** Priority_Specific_Dispatching (
*policy*_identifier, *first_priority*_expression, *last_priority*_expression); — See D.2.2.

27.2/2 **pragma** Profile (*profile*_identifier {, *profile*_pragma_argument_association}); — See D.13.

28 **pragma** Pure[(*library_unit*_name)]; — See 10.2.1.

29 **pragma** Queuing_Policy(*policy*_identifier); — See D.4.

29.1/2 **pragma** Relative_Deadline (*relative_deadline*_expression); — See D.2.6.

30 **pragma** Remote_Call_Interface[(*library_unit*_name)]; — See E.2.3.

31 **pragma** Remote_Types[(*library_unit*_name)]; — See E.2.2.

32 **pragma** Restrictions(restriction{, restriction}); — See 13.12.

33 **pragma** Reviewable; — See H.3.1.

34 **pragma** Shared_Passive[(*library_unit*_name)]; — See E.2.1.

35 **pragma** Storage_Size(expression); — See 13.3.

36 **pragma** Suppress(identifier); — See 11.5.

37 **pragma** Task_Dispatching_Policy(*policy*_identifier); — See D.2.2.

37.1/2 **pragma** Unchecked_Union (*first_subtype*_local_name); — See B.3.3.

37.2/2 **pragma** Unsuppress(identifier); — See 11.5.

38 **pragma** Volatile(local_name); — See C.6.

39 **pragma** Volatile_Components(*array*_local_name); — See C.6.

Wording Changes from Ada 83

39.a Pragmas List, Page, and Optimize are now officially defined in 2.8, “Pragmas”.

Annex M (informative) Summary of Documentation Requirements

{*documentation requirements*} The Ada language allows for certain target machine dependences in a controlled manner. Each Ada implementation must document many characteristics and properties of the target system. This International Standard contains specific documentation requirements. In addition, many characteristics that require documentation are identified throughout this International Standard as being implementation defined. Finally, this International Standard requires documentation of whether implementation advice is followed. The following clauses provide summaries of these documentation requirements. 1/2

M.1 Specific Documentation Requirements

{*documentation requirements (summary of requirements)*} {*documentation (required of an implementation)*} In addition to implementation-defined characteristics, each Ada implementation must document various properties of the implementation: 1/2

> **Ramification:** Most of the items in this list require documentation only for implementations that conform to Specialized Needs Annexes. 1.a/2

- The behavior of implementations in implementation-defined situations shall be documented — see M.2, “Implementation-Defined Characteristics” for a listing. See 1.1.3(19). 2/2
- The set of values that a user-defined Allocate procedure needs to accept for the Alignment parameter. How the standard storage pool is chosen, and how storage is allocated by standard storage pools. See 13.11(22). 3/2
- The algorithm used for random number generation, including a description of its period. See A.5.2(44). 4/2
- The minimum time interval between calls to the time-dependent Reset procedure that is guaranteed to initiate different random number sequences. See A.5.2(45). 5/2
- The conditions under which Io_Exceptions.Name_Error, Io_Exceptions.Use_Error, and Io_Exceptions.Device_Error are propagated. See A.13(15). 6/2
- The behavior of package Environment_Variables when environment variables are changed by external mechanisms. See A.17(30/2). 7/2
- The overhead of calling machine-code or intrinsic subprograms. See C.1(6). 8/2
- The types and attributes used in machine code insertions. See C.1(7). 9/2
- The subprogram calling conventions for all supported convention identifiers. See C.1(8). 10/2
- The mapping between the Link_Name or Ada designator and the external link name. See C.1(9). 11/2
- The treatment of interrupts. See C.3(22). 12/2
- The metrics for interrupt handlers. See C.3.1(16). 13/2
- If the Ceiling_Locking policy is in effect, the default ceiling priority for a protected object that contains an interrupt handler pragma. See C.3.2(24/2). 14/2
- Any circumstances when the elaboration of a preelaborated package causes code to be executed. See C.4(12). 15/2
- Whether a partition can be restarted without reloading. See C.4(13). 16/2

17/2 • The effect of calling Current_Task from an entry body or interrupt handler. See C.7.1(19).

18/2 • For package Task_Attributes, limits on the number and size of task attributes, and how to configure any limits. See C.7.2(19).

19/2 • The metrics for the Task_Attributes package. See C.7.2(27).

20/2 • The details of the configuration used to generate the values of all metrics. See D(2).

21/2 • The maximum priority inversion a user task can experience from the implementation. See D.2.3(12/2).

22/2 • The amount of time that a task can be preempted for processing on behalf of lower-priority tasks. See D.2.3(13/2).

23/2 • The quantum values supported for round robin dispatching. See D.2.5(16/2).

24/2 • The accuracy of the detection of the exhaustion of the budget of a task for round robin dispatching. See D.2.5(17/2).

25/2 • Any conditions that cause the completion of the setting of the deadline of a task to be delayed for a multiprocessor. See D.2.6(32/2).

26/2 • Any conditions that cause the completion of the setting of the priority of a task to be delayed for a multiprocessor. See D.5.1(12.1/2).

27/2 • The metrics for Set_Priority. See D.5.1(14).

28/2 • The metrics for setting the priority of a protected object. See D.5.2(10).

29/2 • On a multiprocessor, any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. See D.6(3).

30/2 • The metrics for aborts. See D.6(8).

31/2 • The values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick for package Real_Time. See D.8(33).

32/2 • The properties of the underlying time base used in package Real_Time. See D.8(34).

33/2 • Any synchronization of package Real_Time with external time references. See D.8(35).

34/2 • Any aspects of the external environment that could interfere with package Real_Time. See D.8(36/1).

35/2 • The metrics for package Real_Time. See D.8(45).

36/2 • The minimum value of the delay expression of a delay_relative_statement that causes a task to actually be blocked. See D.9(7).

37/2 • The minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked. See D.9(8).

38/2 • The metrics for delay statements. See D.9(13).

39/2 • The upper bound on the duration of interrupt blocking caused by the implementation. See D.12(5).

40/2 • The metrics for entry-less protected objects. See D.12(12).

41/2 • The values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick of package Execution_Time. See D.14(21/2).

42/2 • The properties of the mechanism used to implement package Execution_Time. See D.14(22/2).

43/2 • The metrics for execution time. See D.14(27).

44/2 • The metrics for timing events. See D.15(24).

- Whether the RPC-receiver is invoked from concurrent tasks, and if so, the number of such tasks. See E.5(25). 45/2
- Any techniques used to reduce cancellation errors in Numerics.Generic_Real_Arrays shall be documented. See G.3.1(86/2). 46/2
- Any techniques used to reduce cancellation errors in Numerics.Generic_Complex_Arrays shall be documented. See G.3.2(155/2). 47/2
- If a pragma Normalize_Scalars applies, the implicit initial values of scalar subtypes shall be documented. Such a value should be an invalid representation when possible; any cases when is it not shall be documented. See H.1(5/2). 48/2
- The range of effects for each bounded error and each unspecified effect. If the effects of a given erroneous construct are constrained, the constraints shall be documented. See H.2(1). 49/2
- For each inspection point, a mapping between each inspectable object and the machine resources where the object's value can be obtained shall be provided. See H.3.2(8). 50/2
- If a pragma Restrictions(No_Exceptions) is specified, the effects of all constructs where language-defined checks are still performed. See H.4(25). 51/2
- The interrupts to which a task entry may be attached. See J.7.1(12). 52/2
- The type of entry call invoked for an interrupt entry. See J.7.1(13). 53/2

M.2 Implementation-Defined Characteristics

{*implementation defined (summary of characteristics)*} The Ada language allows for certain machine dependences in a controlled manner. {*documentation (required of an implementation)*} Each Ada implementation must document all implementation-defined characteristics: 1/2

Ramification: {*unspecified*} {*specified (not!)*} It need not document unspecified characteristics. 1.a

Some of the items in this list require documentation only for implementations that conform to Specialized Needs Annexes. 1.b

- Whether or not each recommendation given in Implementation Advice is followed — see M.3, “Implementation Advice” for a listing. See 1.1.2(37). 2/2
- Capacity limitations of the implementation. See 1.1.3(3). 3
- Variations from the standard that are impractical to avoid given the implementation's execution environment. See 1.1.3(6). 4
- Which code_statements cause external interactions. See 1.1.3(10). 5
- The semantics of an Ada program whose text is not in Normalization Form KC. See 2.1(4.1/2). 5.1/2
- The coded representation for the text of an Ada program. See 2.1(4/2). 6
- *This paragraph was deleted.* 7/2
- The representation for an end of line. See 2.2(2/2). 8
- Maximum supported line length and lexical element length. See 2.2(14). 9
- Implementation-defined pragmas. See 2.8(14). 10
- Effect of pragma Optimize. See 2.8(27). 11
- The sequence of characters of the value returned by S'Wide_Image when some of the graphic characters of S'Wide_Wide_Image are not defined in Wide_Character. See 3.5(30/2). 11.1/2
- The sequence of characters of the value returned by S'Image when some of the graphic characters of S'Wide_Wide_Image are not defined in Character. See 3.5(37/2). 12/2

13 • The predefined integer types declared in Standard. See 3.5.4(25).

14 • Any nonstandard integer types and the operators defined for them. See 3.5.4(26).

15 • Any nonstandard real types and the operators defined for them. See 3.5.6(8).

16 • What combinations of requested decimal precision and range are supported for floating point types. See 3.5.7(7).

17 • The predefined floating point types declared in Standard. See 3.5.7(16).

18 • The *small* of an ordinary fixed point type. See 3.5.9(8/2).

19 • What combinations of *small*, range, and *digits* are supported for fixed point types. See 3.5.9(10).

19.1/2 • The sequence of characters of the value returned by Tags.Expanded_Name (respectively, Tags.Wide_Expanded_Name) when some of the graphic characters of Tags.Wide_Wide_Expanded_Name are not defined in Character (respectively, Wide_Character). See 3.9(10.1/2).

20/2 • The result of Tags.Wide_Wide_Expanded_Name for types declared within an unnamed block_statement. See 3.9(10).

21 • Implementation-defined attributes. See 4.1.4(12/1).

21.1/2 • Rounding of real static expressions which are exactly half-way between two machine numbers. See 4.9(38/2).

22 • Any implementation-defined time types. See 9.6(6).

23 • The time base associated with relative delays. See 9.6(20).

24 • The time base of the type Calendar.Time. See 9.6(23).

25/2 • The time zone used for package Calendar operations. See 9.6(24/2).

26 • Any limit on delay_until_statements of select_statements. See 9.6(29).

26.1/2 • The result of Calendar.Formating.Image if its argument represents more than 100 hours. See 9.6.1(86/2).

27 • Whether or not two nonoverlapping parts of a composite object are independently addressable, in the case where packing, record layout, or Component_Size is specified for the object. See 9.10(1).

28 • The representation for a compilation. See 10.1(2).

29 • Any restrictions on compilations that contain multiple compilation_units. See 10.1(4).

29.1/2 • The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment. See 10.1.4(3).

30 • The mechanisms for creating an environment and for adding and replacing compilation units. See 10.1.4(3/2).

31 • The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit. See 10.2(2).

32 • The manner of explicitly assigning library units to a partition. See 10.2(2).

33 • The manner of designating the main subprogram of a partition. See 10.2(7).

34 • The order of elaboration of library_items. See 10.2(18).

35 • Parameter passing and function return for the main subprogram. See 10.2(21).

36 • The mechanisms for building and running partitions. See 10.2(24).

37 • The details of program execution, including program termination. See 10.2(25).

- The semantics of any nonactive partitions supported by the implementation. See 10.2(28). 38
- The information returned by Exception_Message. See 11.4.1(10.1/2). 39
- The sequence of characters of the value returned by Exceptions.Exception_Name (respectively, Exceptions.Wide_Exception_Name) when some of the graphic characters of Exceptions.Wide_Wide_Exception_Name are not defined in Character (respectively, Wide_Character). See 11.4.1(12.1/2). 39.1/2
- The result of Exceptions.Wide_Wide_Exception_Name for exceptions declared within an unnamed block_statement. See 11.4.1(12). 40/2
- The information returned by Exception_Information. See 11.4.1(13/2). 41
- Implementation-defined *policy_*identifiers allowed in a pragma Assertion_Policy. See 11.4.2(9/2). 41.1/2
- The default assertion policy. See 11.4.2(10/2). 41.2/2
- Existence and meaning of second parameter of pragma Unsuppress. See 11.5(27.1/2). 41.3/2
- Implementation-defined check names. See 11.5(27). 42
- The cases that cause conflicts between the representation of the ancestors of a type_declaration. See 13.1(13.1/2). 42.1/2
- Any restrictions placed upon representation items. See 13.1(20). 43
- The interpretation of each aspect of representation. See 13.1(20). 44
- The set of machine scalars. See 13.3(8.1/2). 44.1/2
- The meaning of Size for indefinite subtypes. See 13.3(48). 45
- The default external representation for a type tag. See 13.3(75/1). 46
- What determines whether a compilation unit is the same in two different partitions. See 13.3(76). 47
- Implementation-defined components. See 13.5.1(15). 48
- If Word_Size = Storage_Unit, the default bit ordering. See 13.5.3(5). 49
- The contents of the visible part of package System. See 13.7(2). 50/2
- The range of Storage_Elements.Storage_Offset, the modulus of Storage_Elements.Storage_Element, and the declaration of Storage_Elements.Integer_Address.. See 13.7.1(11). 50.1/2
- The contents of the visible part of package System.Machine_Code, and the meaning of code_statements. See 13.8(7). 51
- The effect of unchecked conversion for instances with nonscalar result types whose effect is not defined by the language. See 13.9(11). 52/2
- The result of unchecked conversion for instances with scalar result types whose result is not defined by the language. See 13.9(11). 52.1/2
- Whether or not the implementation provides user-accessible names for the standard pool type(s). See 13.11(17). 53
- *This paragraph was deleted.* 54/2
- The meaning of Storage_Size when neither the Storage_Size nor the Storage_Pool is specified for an access type. See 13.11(18). 55/2
- *This paragraph was deleted.* 56/2
- The set of restrictions allowed in a pragma Restrictions. See 13.12(7/2). 57/2

58 • The consequences of violating limitations on Restrictions pragmas. See 13.12(9).

59/2 • The contents of the stream elements read and written by the Read and Write attributes of elementary types. See 13.13.2(9).

60 • The names and characteristics of the numeric subtypes declared in the visible part of package Standard. See A.1(3).

60.1/2 • The values returned by Strings.Hash. See A.4.9(3/2).

61 • The accuracy actually achieved by the elementary functions. See A.5.1(1).

62 • The sign of a zero result from some of the operators or functions in Numerics.Generic_Elementary_Functions, when Float_Type'Signed_Zeros is True. See A.5.1(46).

63 • The value of Numerics.Discrete_Random.Max_Image_Width. See A.5.2(27).

64 • The value of Numerics.Float_Random.Max_Image_Width. See A.5.2(27).

65/2 • *This paragraph was deleted.*

66 • The string representation of a random number generator's state. See A.5.2(38).

67/2 • *This paragraph was deleted.*

68 • The values of the Model_Mantissa, Model_Emin, Model_Epsilon, Model, Safe_First, and Safe_Last attributes, if the Numerics Annex is not supported. See A.5.3(72).

69/2 • *This paragraph was deleted.*

70 • The value of Buffer_Size in Storage_IO. See A.9(10).

71/2 • The external files associated with the standard input, standard output, and standard error files. See A.10(5).

72 • The accuracy of the value produced by Put. See A.10.9(36).

72.1/1 • Current size for a stream file for which positioning is not supported. See A.12.1(1.1/1).

73/2 • The meaning of Argument_Count, Argument, and Command_Name for package Command_Line. The bounds of type Command_Line.Exit_Status. See A.15(1).

73.1/2 • The interpretation of file names and directory names. See A.16(46/2).

73.2/2 • The maximum value for a file size in Directories. See A.16(87/2).

73.3/2 • The result for Directories.Size for a directory or special file See A.16(93/2).

73.4/2 • The result for Directories.Modification_Time for a directory or special file. See A.16(95/2).

73.5/2 • The interpretation of a non-null search pattern in Directories. See A.16(104/2).

73.6/2 • The results of a Directories search if the contents of the directory are altered while a search is in progress. See A.16(110/2).

73.7/2 • The definition and meaning of an environment variable. See A.17(1/2).

73.8/2 • The circumstances where an environment variable cannot be defined. See A.17(16/2).

73.9/2 • Environment names for which Set has the effect of Clear. See A.17(17/2).

73.10/2 • The value of Containers.Hash_Type'Modulus. The value of Containers.Count_Type'Last. See A.18.1(7/2).

74 • Implementation-defined convention names. See B.1(11).

75 • The manner of choosing link names when neither the link name nor the address of an imported or exported entity is specified. See B.1(36).

- The meaning of link names. See B.1(36). 76
- The effect of pragma Linker_Options. See B.1(37). 77
- The contents of the visible part of package Interfaces and its language-defined descendants. See B.2(1). 78
- Implementation-defined children of package Interfaces. See B.2(11). 79/2
- The definitions of certain types and constants in Interfaces.C. See B.3(41). 79.1/2
- The types Floating, Long_Floating, Binary, Long_Binary, Decimal_Element, and COBOL_Character; and the initializations of the variables Ada_To_COBOL and COBOL_To_Ada, in Interfaces.COBOL. See B.4(50). 80/1
- The types Fortran_Integer, Real, Double_Precision, and Character_Set in Interfaces.Fortran. See B.5(17). 80.1/1
- Implementation-defined intrinsic subprograms. See C.1(1). 81/2
- *This paragraph was deleted.* 82/2
- *This paragraph was deleted.* 83/2
- Any restrictions on a protected procedure or its containing type when a pragma Attach_handler or Interrupt_Handler applies. See C.3.1(17). 83.1/2
- Any other forms of interrupt handler supported by the Attach_Handler and Interrupt_Handler pragmas. See C.3.1(19). 83.2/2
- *This paragraph was deleted.* 84/2
- The semantics of pragma Discard_Names. See C.5(7). 85
- The result of the Task_Identification.Image attribute. See C.7.1(7). 86
- The value of Current_Task when in a protected entry, interrupt handler, or finalization of a task attribute. See C.7.1(17/2). 87/2
- *This paragraph was deleted.* 88/2
- Granularity of locking for Task_Attributes. See C.7.2(16/1). 88.1/1
- *This paragraph was deleted.* 89/2
- *This paragraph was deleted.* 90/2
- The declarations of Any_Priority and Priority. See D.1(11). 91
- Implementation-defined execution resources. See D.1(15). 92
- Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy. See D.2.1(3). 93
- The effect of implementation-defined execution resources on task dispatching. See D.2.1(9/2). 94/2
- *This paragraph was deleted.* 95/2
- *This paragraph was deleted.* 96/2
- Implementation defined task dispatching policies. See D.2.2(18). 97/2
- The value of Default_Quantum in Dispatching.Round_Robin. See D.2.5(4). 97.1/2
- Implementation-defined *policy*_identifiers allowed in a pragma Locking_Policy. See D.3(4). 98
- The locking policy if no Locking_Policy pragma applies to any unit of a partition. See D.3(6). 98.1/2
- Default ceiling priorities. See D.3(10/2). 99

100 • The ceiling of any protected object used internally by the implementation. See D.3(16).

101 • Implementation-defined queuing policies. See D.4(1/1).

102/2 • *This paragraph was deleted.*

103 • Any operations that implicitly require heap storage allocation. See D.7(8).

103.1/2 • When restriction No_Task_Termination applies to a partition, what happens when a task terminates. See D.7(15.1/2).

103.2/2 • The behavior when restriction Max_Storage_At_Blocking is violated. See D.7(17/1).

103.3/2 • The behavior when restriction Max_Asynchronous_Select_Nesting is violated. See D.7(18/1).

103.4/2 • The behavior when restriction Max_Tasks is violated. See D.7(19).

104/2 • Whether the use of pragma Restrictions results in a reduction in program code or data size or execution time. See D.7(20).

105/2 • *This paragraph was deleted.*

106/2 • *This paragraph was deleted.*

107/2 • *This paragraph was deleted.*

108 • The means for creating and executing distributed programs. See E(5).

109 • Any events that can result in a partition becoming inaccessible. See E.1(7).

110 • The scheduling policies, treatment of priorities, and management of shared resources between partitions in certain cases. See E.1(11).

111/1 • *This paragraph was deleted.*

112 • Whether the execution of the remote subprogram is immediately aborted as a result of cancellation. See E.4(13).

112.1/2 • The range of type System.RPC.Partition_Id. See E.5(14).

113/2 • *This paragraph was deleted.*

114 • Implementation-defined interfaces in the PCS. See E.5(26).

115 • The values of named numbers in the package Decimal. See F.2(7).

116 • The value of Max_Picture_Length in the package Text_IO.Editing See F.3.3(16).

117 • The value of Max_Picture_Length in the package Wide_Text_IO.Editing See F.3.4(5).

117.1/2 • The value of Max_Picture_Length in the package Wide_Wide_Text_IO.Editing See F.3.5(5).

118 • The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations. See G.1(1).

119 • The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Types, when Real'Signed_Zeros is True. See G.1.1(53).

120 • The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Elementary_Functions, when Complex_Types.Real'Signed_Zeros is True. See G.1.2(45).

121 • Whether the strict mode or the relaxed mode is the default. See G.2(2).

122 • The result interval in certain cases of fixed-to-float conversion. See G.2.1(10).

123 • The result of a floating point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.1(13).

- The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal. See G.2.1(16). 124
- The definition of *close result set*, which determines the accuracy of certain fixed point multiplications and divisions. See G.2.3(5). 125
- Conditions on a *universal_real* operand of a fixed point multiplication or division for which the result shall be in the *perfect result set*. See G.2.3(22). 126
- The result of a fixed point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.3(27). 127
- The result of an elementary function reference in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.4(4). 128
- The accuracy of certain elementary functions for parameters beyond the angle threshold. See G.2.4(10). 129
- The value of the *angle threshold*, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound. See G.2.4(10). 130
- The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the Machine_Overflows attribute of the corresponding real type is False. See G.2.6(5). 131
- The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold. See G.2.6(8). 132
- The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Real_Matrix. See G.3.1(81/2). 132.1/2
- The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Complex_Matrix. See G.3.2(149/2). 132.2/2
- *This paragraph was deleted.* 133/2
- *This paragraph was deleted.* 134/2
- *This paragraph was deleted.* 135/2
- *This paragraph was deleted.* 136/2
- Implementation-defined *policy_*identifiers allowed in a pragma Partition_Elaboration_Policy. See H.6(4/2). 136.1/2

M.3 Implementation Advice

1/2 {*implementation advice (summary of advice)*} {*documentation (required of an implementation)*} This International Standard sometimes gives advice about handling certain target machine dependences. Each Ada implementation must document whether that advice is followed:

1.a/2 **Ramification:** Some of the items in this list require documentation only for implementations that conform to Specialized Needs Annexes.

- Program_Error should be raised when an unsupported Specialized Needs Annex feature is used at run time. See 1.1.3(20). 2/2
- Implementation-defined extensions to the functionality of a language-defined library unit should be provided by adding children to the library unit. See 1.1.3(21). 3/2
- If a bounded error or erroneous execution is detected, Program_Error should be raised. See 1.1.5(12). 4/2

5/2 • Implementation-defined pragmas should have no semantic effect for error-free programs. See 2.8(16).

6/2 • Implementation-defined pragmas should not make an illegal program legal, unless they complete a declaration or configure the library_items in an environment. See 2.8(19).

7/2 • Long_Integer should be declared in Standard if the target supports 32-bit arithmetic. No other named integer subtypes should be declared in Standard. See 3.5.4(28).

8/2 • For a two's complement target, modular types with a binary modulus up to System.Max_Int*2+2 should be supported. A nonbinary modulus up to Integer'Last should be supported. See 3.5.4(29).

9/2 • Program_Error should be raised for the evaluation of S'Pos for an enumeration type, if the value of the operand does not correspond to the internal code for any enumeration literal of the type. See 3.5.5(8).

10/2 • Long_Float should be declared in Standard if the target supports 11 or more digits of precision. No other named float subtypes should be declared in Standard. See 3.5.7(17).

11/2 • Multidimensional arrays should be represented in row-major order, unless the array has convention Fortran. See 3.6.2(11).

12/2 • Tags.Internal_Tag should return the tag of a type whose innermost master is the master of the point of the function call.. See 3.9(26.1/2).

13/2 • For a real static expression with a non-formal type that is not part of a larger static expression should be rounded the same as the target system. See 4.9(38.1/2).

14/2 • The value of Duration'Small should be no greater than 100 microseconds. See 9.6(30).

15/2 • The time base for delay_relative_statements should be monotonic. See 9.6(31).

16/2 • Leap seconds should be supported if the target system supports them. Otherwise, operations in Calendar.Formatting should return results consistent with no leap seconds. See 9.6.1(89/2).

17/2 • When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance. See 10.1.5(10/1).

18/2 • A type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package. See 10.2.1(12).

19/2 • Exception_Message by default should be short, provide information useful for debugging, and should not include the Exception_Name. See 11.4.1(19).

20/2 • Exception_Information should provide information useful for debugging, and should include the Exception_Name and Exception_Message. See 11.4.1(19).

21/2 • Code executed for checks that have been suppressed should be minimized. See 11.5(28).

22/2 • The recommended level of support for all representation items should be followed. See 13.1(28/2).

23/2 • Storage allocated to objects of a packed type should be minimized. See 13.2(6).

24/2 • The recommended level of support for pragma Pack should be followed. See 13.2(9).

25/2 • For an array X, X'Address should point at the first component of the array rather than the array bounds. See 13.3(14).

26/2 • The recommended level of support for the Address attribute should be followed. See 13.3(19).

27/2 • The recommended level of support for the Alignment attribute should be followed. See 13.3(35).

28/2 • The Size of an array object should not include its bounds. See 13.3(41.1/2).

- If the Size of a subtype allows for efficient independent addressability, then the Size of most objects of the subtype should equal the Size of the subtype. See 13.3(52). 29/2
- A Size clause on a composite subtype should not affect the internal layout of components. See 13.3(53). 30/2
- The recommended level of support for the Size attribute should be followed. See 13.3(56). 31/2
- The recommended level of support for the Component_Size attribute should be followed. See 13.3(73). 32/2
- The recommended level of support for enumeration_representation_clauses should be followed. See 13.4(10). 33/2
- The recommended level of support for record_representation_clauses should be followed. See 13.5.1(22). 34/2
- If a component is represented using a pointer to the actual data of the component which is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes. See 13.5.2(5). 35/2
- The recommended level of support for the nondefault bit ordering should be followed. See 13.5.3(8). 36/2
- Type System.Address should be a private type. See 13.7(37). 37/2
- Operations in System and its children should reflect the target environment; operations that do not make sense should raise Program_Error. See 13.7.1(16). 38/2
- Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data in an instance of Unchecked_Conversion. See 13.9(14/2). 39/2
- There should not be unnecessary run-time checks on the result of an Unchecked_Conversion; the result should be returned by reference when possible. Restrictions on Unchecked_Conversions should be avoided. See 13.9(15). 40/2
- The recommended level of support for Unchecked_Conversion should be followed. See 13.9(17). 41/2
- Any cases in which heap storage is dynamically allocated other than as part of the evaluation of an allocator should be documented. See 13.11(23). 42/2
- A default storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects. See 13.11(24). 43/2
- Usually, a storage pool for an access discriminant or access parameter should be created at the point of an allocator, and be reclaimed when the designated object becomes inaccessible. For other anonymous access types, the pool should be created at the point where the type is elaborated and need not support deallocation of individual objects. See 13.11(25). 44/2
- For a standard storage pool, an instance of Unchecked_Deallocation should actually reclaim the storage. See 13.11.2(17). 45/2
- The recommended level of support for the Stream_Size attribute should be followed. See 13.13.2(1.8/2). 46/2
- If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size. See 13.13.2(1.6/2). 47/2
- If an implementation provides additional named predefined integer types, then the names should end with “Integer”. If an implementation provides additional named predefined floating point types, then the names should end with “Float”. See A.1(52). 48/2

49/2 • Implementation-defined operations on Wide_Character, Wide_String, Wide_Wide_Character, and Wide_Wide_String should be child units of Wide_Characters or Wide_Wide_Characters. See A.3.1(7/2).

50/2 • Bounded string objects should not be implemented by implicit pointers and dynamic allocation. See A.4.4(106).

51/2 • Strings.Hash should be good a hash function, returning a wide spread of values for different string values, and similar strings should rarely return the same value. See A.4.9(12/2).

52/2 • Any storage associated with an object of type Generator of the random number packages should be reclaimed on exit from the scope of the object. See A.5.2(46).

53/2 • Each value of Initiator passed to Reset for the random number packages should initiate a distinct sequence of random numbers, or, if that is not possible, be at least a rapidly varying function of the initiator value. See A.5.2(47).

54/2 • Get_Immediate should be implemented with unbuffered input; input should be available immediately; line-editing should be disabled. See A.10.7(23).

55/2 • Package Directories.Information should be provided to retrieve other information about a file. See A.16(124/2).

56/2 • Directories.Start_Search and Directories.Search should raise Use_Error for malformed patterns. See A.16(125/2).

57/2 • Directories.Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file. See A.16(126/2).

58/2 • If the execution environment supports subprocesses, the current environment variables should be used to initialize the environment variables of a subprocess. See A.17(32/2).

59/2 • Changes to the environment variables made outside the control of Environment_Variables should be reflected immediately. See A.17(33/2).

60/2 • Containers.Hash_Type'Modulus should be at least 2**32. Containers.Count_Type'Last should be at least 2**31–1. See A.18.1(8/2).

61/2 • The worst-case time complexity of Element for Containers.Vector should be *O*(log *N*). See A.18.2(256/2).

62/2 • The worst-case time complexity of Append with Count = 1 when *N* is less than the capacity for Containers.Vector should be *O*(log *N*). See A.18.2(257).

63/2 • The worst-case time complexity of Prepend with Count = 1 and Delete_First with Count=1 for Containers.Vectors should be *O*(*N* log *N*). See A.18.2(258/2).

64/2 • The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2). See A.18.2(259/2).

65/2 • Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements. See A.18.2(260/2).

66/2 • Containers.Vectors.Move should not copy elements, and should minimize copying of internal data structures. See A.18.2(261/2).

67/2 • If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation. See A.18.2(262/2).

68/2 • The worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 for Containers.Doubly_Linked_Lists should be *O*(log *N*). See A.18.3(160/2).

- a call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). See A.18.3(161/2). 69/2
- Containers.Doubly_Link_Lists.Move should not copy elements, and should minimize copying of internal data structures. See A.18.3(162/2). 70/2
- If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation. See A.18.3(163/2). 71/2
- Move for a map should not copy elements, and should minimize copying of internal data structures. See A.18.4(83/2). 72/2
- If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation. See A.18.4(84/2). 73/2
- The average time complexity of Element, Insert, Include, Replace, Delete, Exclude and Find operations that take a key parameter for Containers.Hashed_Maps should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Maps that take a cursor parameter should be *O*(1). See A.18.5(62/2). 74/2
- The worst-case time complexity of Element, Insert, Include, Replace, Delete, Exclude and Find operations that take a key parameter for Containers.Ordered_Maps should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms of Containers.Ordered_Maps that take a cursor parameter should be *O*(1). See A.18.6(95/2). 75/2
- Move for sets should not copy elements, and should minimize copying of internal data structures. See A.18.7(104/2). 76/2
- If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation. See A.18.7(105/2). 77/2
- The average time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations of Containers.Hashed_Sets that take an element parameter should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Sets that take a cursor parameter should be *O*(1). The average time complexity of Containers.Hashed_Sets.Reserve_Capacity should be *O*(*N*). See A.18.8(88/2). 78/2
- The worst-case time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations of Containers.Ordered_Sets that take an element parameter should be *O*((log *N*)**2). The worst-case time complexity of the subprograms of Containers.Ordered_Sets that take a cursor parameter should be *O*(1). See A.18.9(116/2). 79/2
- Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). See A.18.16(10/2). 80/2
- Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements. See A.18.16(11/2). 81/2
- If pragma Export is supported for a language, the main program should be able to be written in that language. Subprograms named "adainit" and "adafinal" should be provided for elaboration and finalization of the environment task. See B.1(39). 82/2
- Automatic elaboration of preelaborated packages should be provided when pragma Export is supported. See B.1(40). 83/2
- For each supported convention *L* other than Intrinsic, pragmas Import and Export should be supported for objects of *L*-compatible types and for subprograms, and pragma Convention should be supported for *L*-eligible types and for subprograms. See B.1(41). 84/2

85/2 • If an interface to C, COBOL, or Fortran is provided, the corresponding package or packages described in Annex B, “Interface to Other Languages” should also be provided. See B.2(13).

86/2 • The constants nul, wide_nul, char16_nul, and char32_nul in package Interfaces.C should have a representation of zero. See B.3(62.1/2).

87/2 • If C interfacing is supported, the interface correspondences between Ada and C should be supported. See B.3(71).

88/2 • If COBOL interfacing is supported, the interface correspondences between Ada and COBOL should be supported. See B.4(98).

89/2 • If Fortran interfacing is supported, the interface correspondences between Ada and Fortran should be supported. See B.5(26).

90/2 • The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment. See C.1(3).

91/2 • Interface to assembler should be supported; the default assembler should be associated with the convention identifier Assembler. See C.1(4).

92/2 • If an entity is exported to assembly language, then the implementation should allocate it at an addressable location even if not otherwise referenced from the Ada code. A call to a machine code or assembler subprogram should be treated as if it could read or update every object that is specified as exported. See C.1(5).

93/2 • Little or no overhead should be associated with calling intrinsic and machine-code subprograms. See C.1(10).

94/2 • Intrinsic subprograms should be provided to access any machine operations that provide special capabilities or efficiency not normally available. See C.1(16).

95/2 • If the Ceiling_Locking policy is not in effect and the target system allows for finer-grained control of interrupt blocking, a means for the application to specify which interrupts are to be blocked during protected actions should be provided. See C.3(28/2).

96/2 • Interrupt handlers should be called directly by the hardware. See C.3.1(20).

97/2 • Violations of any implementation-defined restrictions on interrupt handlers should be detected before run time. See C.3.1(21).

98/2 • If implementation-defined forms of interrupt handler procedures are supported, then for each such form of a handler, a type analogous to Parameterless_Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts. See C.3.2(25).

99/2 • Preelaborated packages should be implemented such that little or no code is executed at run time for the elaboration of entities. See C.4(14).

100/2 • If **pragma** Discard_Names applies to an entity, then the amount of storage used for storing names associated with that entity should be reduced. See C.5(8).

101/2 • A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others. See C.6(22/2).

102/2 • A load or store of an atomic object should be implemented by a single load or store instruction. See C.6(23/2).

103/2 • Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination. See C.7.2(30.1/2).

- If the target domain requires deterministic memory use at run time, storage for task attributes should be pre-allocated statically and the number of attributes pre-allocated should be documented. See C.7.2(30). 104/2
- Names that end with "_Locking" should be used for implementation-defined locking policies. See D.3(17). 105/2
- Names that end with "_Queuing" should be used for implementation-defined queuing policies. See D.4(16). 106/2
- The abort_statement should not require the task executing the statement to block. See D.6(9). 107/2
- On a multi-processor, the delay associated with aborting a task on another processor should be bounded. See D.6(10). 108/2
- When feasible, specified restrictions should be used to produce a more efficient implementation. See D.7(21). 109/2
- When appropriate, mechanisms to change the value of Tick should be provided. See D.8(47). 110/2
- Calendar.Clock and Real_Time.Clock should be transformations of the same time base. See D.8(48). 111/2
- The "best" time base which exists in the underlying system should be available to the application through Real_Time.Clock. See D.8(49). 112/2
- When appropriate, implementations should provide configuration mechanisms to change the value of Execution_Time.CPU_Tick. See D.14(29/2). 113/2
- For a timing event, the handler should be executed directly by the real-time clock interrupt mechanism. See D.15(25). 114/2
- The PCS should allow for multiple tasks to call the RPC-receiver. See E.5(28). 115/2
- The System.RPC.Write operation should raise Storage_Error if it runs out of space when writing an item. See E.5(29). 116/2
- If COBOL (respectively, C) is supported in the target environment, then interfacing to COBOL (respectively, C) should be supported as specified in Annex B. See F(7). 117/2
- Packed decimal should be used as the internal representation for objects of subtype *S* when *S*'Machine_Radix = 10. See F.1(2). 118/2
- If Fortran (respectively, C) is supported in the target environment, then interfacing to Fortran (respectively, C) should be supported as specified in Annex B. See G(7). 119/2
- Mixed real and complex operations (as well as pure-imaginary and complex operations) should not be performed by converting the real (resp. pure-imaginary) operand to complex. See G.1.1(56). 120/2
- If Real'Signed_Zeros is true for Numerics.Generic_Complex_Types, a rational treatment of the signs of zero results and result components should be provided. See G.1.1(58). 121/2
- If Complex_Types.Real'Signed_Zeros is true for Numerics.Generic_Complex_Elementary_-Functions, a rational treatment of the signs of zero results and result components should be provided. See G.1.2(49). 122/2
- For elementary functions, the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter. Log without a Base parameter should not be implemented by calling Log with a Base parameter. See G.2.4(19). 123/2
- For complex arithmetic, the Compose_From_Polar function without a Cycle parameter should not be implemented by calling Compose_From_Polar with a Cycle parameter. See G.2.6(15). 124/2

125/2 • Solve and Inverse for Numerics.Generic_Real_Arrays should be implemented using established techniques such as LU decomposition and the result should be refined by an iteration on the residuals. See G.3.1(88/2).

126/2 • The equality operator should be used to test that a matrix in Numerics.Generic_Real_Matrix is symmetric. See G.3.1(90/2).

127/2 • Solve and Inverse for Numerics.Generic_Complex_Arrays should be implemented using established techniques and the result should be refined by an iteration on the residuals. See G.3.2(158/2).

128/2 • The equality and negation operators should be used to test that a matrix is Hermitian. See G.3.2(160/2).

129/2 • Mixed real and complex operations should not be performed by converting the real operand to complex. See G.3.2(161/2).

130/2 • The information produced by pragma Reviewable should be provided in both a human-readable and machine-readable form, and the latter form should be documented. See H.3.1(19).

131/2 • Object code listings should be provided both in a symbolic format and in a numeric format. See H.3.1(20).

132/2 • If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration then the partition should be immediately terminated. See H.6(15/2).

Annex N
(informative)
Glossary

{*AI95-00437-01*} {*Glossary*} This Annex contains informal descriptions of some of the terms used in this International Standard. The index provides references to more formal definitions of all of the terms used in this International Standard. 1/2

{*abstract type*} **Abstract type.** An abstract type is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own. 1.1/2

{*access type*} **Access type.** An access type has values that designate aliased objects. Access types correspond to "pointer types" or "reference types" in some other languages. 2

{*aliased*} **Aliased.** An aliased view of an object is one that can be designated by an access value. Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word **aliased**. The Access attribute can be used to create an access value designating an aliased object. 3

{*ancestor*} **Ancestor.** An ancestor of a type is the type itself or, in the case of a type derived from other types, its parent type or one of its progenitor types or one of their ancestors. Note that ancestor and descendant are inverse relationships. 3.1/2

{*array type*} **Array type.** An array type is a composite type whose components are all of the same type. Components are selected by indexing. 4

{*category (of types)*} **Category (of types).** A category of types is a set of types with one or more common properties, such as primitive operations. A category of types that is closed under derivation is also known as a *class*. 4.1/2

{*character type*} **Character type.** A character type is an enumeration type whose values include characters. 5

{*class (of types)*} **Class (of types).** {*closed under derivation*} A class is a set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class. The set of types of a class share common properties, such as their primitive operations. 6/2

{*compilation unit*} **Compilation unit.** The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation_units. A compilation_unit contains either the declaration, the body, or a renaming of a program unit. 7

{*composite type*} **Composite type.** A composite type may have components. 8/2

{*construct*} **Construct.** A *construct* is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under "Syntax". 9

{*controlled type*} **Controlled type.** A controlled type supports user-defined assignment and finalization. Objects are always finalized before being destroyed. 10

{*declaration*} **Declaration.** A *declaration* is a language construct that associates a name with (a view of) an entity. {*explicit declaration*} {*implicit declaration*} A declaration may appear explicitly in the program text (an *explicit* declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an *implicit* declaration). 11

This paragraph was deleted.{*definition*} 12/2

13/2 {*derived type*} **Derived type.** A derived type is a type defined in terms of one or more other types given in a derived type definition. The first of those types is the parent type of the derived type and any others are progenitor types. Each class containing the parent type or a progenitor type also contains the derived type. The derived type inherits properties such as components and primitive operations from the parent and progenitors. A type together with the types derived from it (directly or indirectly) form a derivation class.

13.1/2 {*descendant*} **Descendant.** A type is a descendant of itself, its parent and progenitor types, and their ancestors. Note that descendant and ancestor are inverse relationships.

14 {*discrete type*} **Discrete type.** A discrete type is either an integer type or an enumeration type. Discrete types may be used, for example, in case_statements and as array indices.

15/2 {*discriminant*} **Discriminant.** A discriminant is a parameter for a composite type. It can control, for example, the bounds of a component of the type if the component is an array. A discriminant for a task type can be used to pass data to a task of the type upon creation.

15.1/2 {*elaboration*} **Elaboration.** The process by which a declaration achieves its run-time effect is called elaboration. Elaboration is one of the forms of execution.

16 {*elementary type*} **Elementary type.** An elementary type does not have components.

17 {*enumeration type*} **Enumeration type.** An enumeration type is defined by an enumeration of its values, which may be named by identifiers or character literals.

17.1/2 {*evaluation*} **Evaluation.** The process by which an expression achieves its run-time effect is called evaluation. Evaluation is one of the forms of execution.

18 {*exception*} **Exception.** An *exception* represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an *exception occurrence*. {*raise (an exception)* [partial]} To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. {*handle (an exception)* [partial]} Performing some actions in response to the arising of an exception is called *handling* the exception.

19 {*execution*} **Execution.** The process by which a construct achieves its run-time effect is called *execution*. {*elaboration*} {*evaluation*} Execution of a declaration is also called *elaboration*. Execution of an expression is also called *evaluation*.

19.1/2 {*function*} **Function.** A function is a form of subprogram that returns a result and can be called as part of an expression.

20 {*generic unit*} **Generic unit.** A generic unit is a template for a (nongeneric) program unit; the template can be parameterized by objects, types, subprograms, and packages. An instance of a generic unit is created by a generic_instantiation. The rules of the language are enforced when a generic unit is compiled, using a generic contract model; additional checks are performed upon instantiation to verify the contract is met. That is, the declaration of a generic unit represents a contract between the body of the generic and instances of the generic. Generic units can be used to perform the role that macros sometimes play in other languages.

20.1/2 {*incomplete type*} **Incomplete type.** An incomplete type gives a view of a type that reveals only some of its properties. The remaining properties are provided by the full view given elsewhere. Incomplete types can be used for defining recursive data structures.

21 {*integer type*} **Integer type.** Integer types comprise the signed integer types and the modular types. A signed integer type has a base range that includes both positive and negative numbers, and has operations that may raise an exception when the result is outside the base range. A modular type has a base range

whose lower bound is zero, and has operations with “wraparound” semantics. Modular types subsume what are called “unsigned types” in some other languages.

21.1/2 {*interface type*} **Interface type.** An interface type is a form of abstract tagged type which has no components or concrete operations except possibly null procedures. Interface types are used for composing other interfaces and tagged types and thereby provide multiple inheritance. Only an interface type can be used as a progenitor of another type.

22 {*library unit*} **Library unit.** A library unit is a separately compiled program unit, and is always a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. {*subsystem*} A root library unit, together with its children and grandchildren and so on, form a *subsystem*.

23/2 {*limited type*} **Limited type.** A limited type is a type for which copying (such as in an assignment_statement) is not allowed. A nonlimited type is a type for which copying is allowed.

24 {*object*} **Object.** An object is either a constant or a variable. An object contains a value. An object is created by an object_declaration or by an allocator. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

24.1/2 {*overriding operation*} **Overriding operation.** An overriding operation is one that replaces an inherited primitive operation. Operations may be marked explicitly as overriding or not overriding.

25 {*package*} **Package.** Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.

25.1/2 {*parent*} **Parent.** The parent of a derived type is the first type given in the definition of the derived type. The parent can be almost any kind of type, including an interface type.

26 {*partition*} **Partition.** A *partition* is a part of a program. Each partition consists of a set of library units. Each partition may run in a separate address space, possibly on a separate computer. A program may contain just one partition. A distributed program typically contains multiple partitions, which can execute concurrently.

27 {*pragma*} **Pragma.** A pragma is a compiler directive. There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation-defined) pragmas.

28 {*primitive operations*} **Primitive operations.** The primitive operations of a type are the operations (such as subprograms) declared together with the type declaration. They are inherited by other types in the same class of types. For a tagged type, the primitive subprograms are dispatching subprograms, providing run-time polymorphism. A dispatching subprogram may be called with statically tagged operands, in which case the subprogram body invoked is determined at compile time. Alternatively, a dispatching subprogram may be called using a dispatching call, in which case the subprogram body invoked is determined at run time.

29/2 {*private extension*} **Private extension.** A private extension is a type that extends another type, with the additional properties hidden from its clients.

30/2 {*private type*} **Private type.** A private type gives a view of a type that reveals only some of its properties. The remaining properties are provided by the full view given elsewhere. Private types can be used for defining abstractions that hide unnecessary details from their clients.

30.1/2 {*procedure*} **Procedure.** A procedure is a form of subprogram that does not return a result and can only be called by a statement.

30.2/2 {*progenitor*} **Progenitor.** A progenitor of a derived type is one of the types given in the definition of the derived type other than the first. A progenitor is always an interface type. Interfaces, tasks, and protected types may also have progenitors.

31 {*program*} **Program.** A *program* is a set of *partitions*, each of which may execute in a separate address space, possibly on a separate computer. A partition consists of a set of library units.

32 {*program unit*} **Program unit.** A *program unit* is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units.

33/2 {*protected type*} **Protected type.** A protected type is a composite type whose components are accessible only through one of its protected operations which synchronize concurrent access by multiple tasks.

34 {*real type*} **Real type.** A real type has values that are approximations of the real numbers. Floating point and fixed point types are real types.

35 {*record extension*} **Record extension.** A record extension is a type that extends another type by adding additional components.

36 {*record type*} **Record type.** A record type is a composite type consisting of zero or more named components, possibly of different types.

36.1/2 {*renaming*} **Renaming.** A renaming_declaration is a declaration that does not define a new entity, but instead defines a view of an existing entity.

37 {*scalar type*} **Scalar type.** A scalar type is either a discrete type or a real type.

37.1/2 {*subprogram*} **Subprogram.** A subprogram is a section of a program that can be executed in various contexts. It is invoked by a subprogram call that may qualify the effect of the subprogram through the passing of parameters. There are two forms of subprograms: functions, which return values, and procedures, which do not.

38/2 {*subtype*} **Subtype.** A subtype is a type together with a constraint or null exclusion, which constrains the values of the subtype to satisfy a certain condition. The values of a subtype are a subset of the values of its type.

38.1/2 {*synchronized*} **Synchronized.** A synchronized entity is one that will work safely with multiple tasks at one time. A synchronized interface can be an ancestor of a task or a protected type. Such a task or protected type is called a synchronized tagged type.

39 {*tagged type*} **Tagged type.** The objects of a tagged type have a run-time type tag, which indicates the specific type with which the object was originally created. An operand of a class-wide tagged type can be used in a dispatching call; the tag indicates which subprogram body to invoke. Nondispatching calls, in which the subprogram body to invoke is determined at compile time, are also allowed. Tagged types may be extended with additional components.

40/2 {*task type*} **Task type.** A task type is a composite type used to represent active entities which execute concurrently and which can communicate via queued task entries. The top-level task of a partition is called the environment task.

{*type*} **Type.** Each object has a type. A *type* has an associated set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. Types are grouped into *categories*. Most language-defined categories of types are also *classes* of types. 41/2

{*view*} **View.** A view of an entity reveals some or all of the properties of the entity. A single entity may have multiple views. 42/2

Annex P
(informative)
Syntax Summary

{*syntax (complete listing)*} {*grammar (complete listing)*} {*context free grammar (complete listing)*} {*BNF (Backus-Naur Form) (complete listing)*} {*Backus-Naur Form (BNF) (complete listing)*} This Annex summarizes the complete syntax of the language. See 1.1.4 for a description of the notation used.

```
2.3:
identifier ::=
  identifier_start {identifier_start | identifier_extend}

2.3:
identifier_start ::=
    letter_uppercase
  | letter_lowercase
  | letter_titlecase
  | letter_modifier
  | letter_other
  | number_letter

2.3:
identifier_extend ::=
    mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector
  | other_format

2.4:
numeric_literal ::= decimal_literal | based_literal

2.4.1:
decimal_literal ::= numeral [.numeral] [exponent]

2.4.1:
numeral ::= digit {[underline] digit}

2.4.1:
exponent ::= E [+] numeral | E – numeral

2.4.1:
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

2.4.2:
based_literal ::=
  base # based_numeral [.based_numeral] # [exponent]

2.4.2:
base ::= numeral

2.4.2:
based_numeral ::=
  extended_digit {[underline] extended_digit}

2.4.2:
extended_digit ::= digit | A | B | C | D | E | F

2.5:
character_literal ::= 'graphic_character'

2.6:
string_literal ::= "{string_element}"

2.6:
string_element ::= "" | non_quotation_mark_graphic_character

2.7:
comment ::= --{non_end_of_line_character}
```

2.8:
pragma ::=
  pragma identifier [(pragma_argument_association {, pragma_argument_association})];

2.8:
pragma_argument_association ::=
    [*pragma_argument_*identifier =>] name
  | [*pragma_argument_*identifier =>] expression

3.1:
basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | null_procedure_declaration | package_declaration
  | renaming_declaration | exception_declaration
  | generic_declaration | generic_instantiation

3.1:
defining_identifier ::= identifier

3.2.1:
type_declaration ::= full_type_declaration
  | incomplete_type_declaration
  | private_type_declaration
  | private_extension_declaration

3.2.1:
full_type_declaration ::=
    type defining_identifier [known_discriminant_part] **is** type_definition;
  | task_type_declaration
  | protected_type_declaration

3.2.1:
type_definition ::=
    enumeration_type_definition | integer_type_definition
  | real_type_definition | array_type_definition
  | record_type_definition | access_type_definition
  | derived_type_definition | interface_type_definition

3.2.2:
subtype_declaration ::=
  subtype defining_identifier **is** subtype_indication;

3.2.2:
subtype_indication ::= [null_exclusion] subtype_mark [constraint]

3.2.2:
subtype_mark ::= *subtype_*name

3.2.2:
constraint ::= scalar_constraint | composite_constraint

3.2.2:
scalar_constraint ::=
    range_constraint | digits_constraint | delta_constraint

3.2.2:
composite_constraint ::=
    index_constraint | discriminant_constraint

3.3.1:
object_declaration ::=
    defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression];
  | defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression];
  | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression];
  | single_task_declaration
  | single_protected_declaration

3.3.1:
defining_identifier_list ::=
  defining_identifier {, defining_identifier}

3.3.2:
number_declaration ::=
  defining_identifier_list : **constant** := *static*_expression;

3.4:
derived_type_definition ::=
  [**abstract**] [**limited**] **new** *parent*_subtype_indication [[**and** interface_list] record_extension_part]

3.5:
range_constraint ::= **range** range

3.5:
range ::= range_attribute_reference
  | simple_expression .. simple_expression

3.5.1:
enumeration_type_definition ::=
  (enumeration_literal_specification {, enumeration_literal_specification})

3.5.1:
enumeration_literal_specification ::= defining_identifier | defining_character_literal

3.5.1:
defining_character_literal ::= character_literal

3.5.4:
integer_type_definition ::= signed_integer_type_definition | modular_type_definition

3.5.4:
signed_integer_type_definition ::= **range** *static*_simple_expression .. *static*_simple_expression

3.5.4:
modular_type_definition ::= **mod** *static*_expression

3.5.6:
real_type_definition ::=
  floating_point_definition | fixed_point_definition

3.5.7:
floating_point_definition ::=
  digits *static*_expression [real_range_specification]

3.5.7:
real_range_specification ::=
  range *static*_simple_expression .. *static*_simple_expression

3.5.9:
fixed_point_definition ::= ordinary_fixed_point_definition | decimal_fixed_point_definition

3.5.9:
ordinary_fixed_point_definition ::=
  delta *static*_expression real_range_specification

3.5.9:
decimal_fixed_point_definition ::=
  delta *static*_expression **digits** *static*_expression [real_range_specification]

3.5.9:
digits_constraint ::=
  digits *static*_expression [range_constraint]

3.6:
array_type_definition ::=
  unconstrained_array_definition | constrained_array_definition

3.6:
unconstrained_array_definition ::=
  array(index_subtype_definition {, index_subtype_definition}) **of** component_definition

3.6:
index_subtype_definition ::= subtype_mark **range** <>

3.6:
constrained_array_definition ::=
  array (discrete_subtype_definition {, discrete_subtype_definition}) **of** component_definition

3.6:
discrete_subtype_definition ::= *discrete*_subtype_indication | range

3.6:
component_definition ::=
  [**aliased**] subtype_indication
 | [**aliased**] access_definition

3.6.1:
index_constraint ::= (discrete_range {, discrete_range})

3.6.1:
discrete_range ::= *discrete*_subtype_indication | range

3.7:
discriminant_part ::= unknown_discriminant_part | known_discriminant_part

3.7:
unknown_discriminant_part ::= (<>)

3.7:
known_discriminant_part ::=
  (discriminant_specification {; discriminant_specification})

3.7:
discriminant_specification ::=
  defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]

3.7:
default_expression ::= expression

3.7.1:
discriminant_constraint ::=
  (discriminant_association {, discriminant_association})

3.7.1:
discriminant_association ::=
  [*discriminant*_selector_name {| *discriminant*_selector_name} =>] expression

3.8:
record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition

3.8:
record_definition ::=
   record
     component_list
   end record
  | **null record**

3.8:
component_list ::=
    component_item {component_item}
  | {component_item} variant_part
  | **null**;

3.8:
component_item ::= component_declaration | aspect_clause

3.8:
component_declaration ::=
  defining_identifier_list : component_definition [:= default_expression];

3.8.1:
variant_part ::=
  case *discriminant*_direct_name **is**
     variant
    {variant}
  end case;

3.8.1:
variant ::=
  when discrete_choice_list =>
    component_list

3.8.1:
discrete_choice_list ::= discrete_choice {| discrete_choice}

3.8.1:
discrete_choice ::= expression | discrete_range | **others**

3.9.1:
record_extension_part ::= **with** record_definition

3.9.3:
abstract_subprogram_declaration ::=
   [overriding_indicator]
   subprogram_specification **is abstract**;

3.9.4:
interface_type_definition ::=
   [**limited** | **task** | **protected** | **synchronized**] **interface** [**and** interface_list]

3.9.4:
interface_list ::= *interface*_subtype_mark {**and** *interface*_subtype_mark}

3.10:
access_type_definition ::=
   [null_exclusion] access_to_object_definition
 | [null_exclusion] access_to_subprogram_definition

3.10:
access_to_object_definition ::=
   access [general_access_modifier] subtype_indication

3.10:
general_access_modifier ::= **all** | **constant**

3.10:
access_to_subprogram_definition ::=
   access [**protected**] **procedure** parameter_profile
 | **access** [**protected**] **function** parameter_and_result_profile

3.10:
null_exclusion ::= **not null**

3.10:
access_definition ::=
   [null_exclusion] **access** [**constant**] subtype_mark
 | [null_exclusion] **access** [**protected**] **procedure** parameter_profile
 | [null_exclusion] **access** [**protected**] **function** parameter_and_result_profile

3.10.1:
incomplete_type_declaration ::= **type** defining_identifier [discriminant_part] [**is tagged**];

3.11:
declarative_part ::= {declarative_item}

3.11:
declarative_item ::=
   basic_declarative_item | body

3.11:
basic_declarative_item ::=
   basic_declaration | aspect_clause | use_clause

3.11:
body ::= proper_body | body_stub

3.11:
proper_body ::=
   subprogram_body | package_body | task_body | protected_body

4.1:
name ::=
     direct_name | explicit_dereference
   | indexed_component | slice
   | selected_component | attribute_reference
   | type_conversion | function_call
   | character_literal

4.1:
direct_name ::= identifier | operator_symbol

4.1:
prefix ::= name | implicit_dereference

4.1:
explicit_dereference ::= name.**all**

4.1:
implicit_dereference ::= name

4.1.1:
indexed_component ::= prefix(expression {, expression})

4.1.2:
slice ::= prefix(discrete_range)

4.1.3:
selected_component ::= prefix . selector_name

4.1.3:
selector_name ::= identifier | character_literal | operator_symbol

4.1.4:
attribute_reference ::= prefix'attribute_designator

4.1.4:
attribute_designator ::=
    identifier[(*static*_expression)]
  | Access | Delta | Digits

4.1.4:
range_attribute_reference ::= prefix'range_attribute_designator

4.1.4:
range_attribute_designator ::= Range[(*static*_expression)]

4.3:
aggregate ::= record_aggregate | extension_aggregate | array_aggregate

4.3.1:
record_aggregate ::= (record_component_association_list)

4.3.1:
record_component_association_list ::=
    record_component_association {, record_component_association}
  | **null record**

4.3.1:
record_component_association ::=
    [component_choice_list =>] expression
   | component_choice_list => <>

4.3.1:
component_choice_list ::=
     *component*_selector_name {| *component*_selector_name}
   | **others**

4.3.2:
extension_aggregate ::=
    (ancestor_part **with** record_component_association_list)

4.3.2:
ancestor_part ::= expression | subtype_mark

```
4.3.3:
array_aggregate ::=
  positional_array_aggregate | named_array_aggregate

4.3.3:
positional_array_aggregate ::=
    (expression, expression {, expression})
  | (expression {, expression}, others => expression)
  | (expression {, expression}, others => <>)

4.3.3:
named_array_aggregate ::=
    (array_component_association {, array_component_association})

4.3.3:
array_component_association ::=
    discrete_choice_list => expression
  | discrete_choice_list => <>

4.4:
expression ::=
     relation {and relation}    | relation {and then relation}
   | relation {or relation}     | relation {or else relation}
   | relation {xor relation}

4.4:
relation ::=
     simple_expression [relational_operator simple_expression]
   | simple_expression [not] in range
   | simple_expression [not] in subtype_mark

4.4:
simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}

4.4:
term ::= factor {multiplying_operator factor}

4.4:
factor ::= primary [** primary] | abs primary | not primary

4.4:
primary ::=
   numeric_literal | null | string_literal | aggregate
 | name | qualified_expression | allocator | (expression)

4.5:
logical_operator ::=                  and | or  | xor

4.5:
relational_operator ::=               =  | /=  | <  | <= | > | >=

4.5:
binary_adding_operator ::=            +  | –  | &

4.5:
unary_adding_operator ::=             +  | –

4.5:
multiplying_operator ::=              *  | /  | mod | rem

4.5:
highest_precedence_operator ::=       **  | abs | not

4.6:
type_conversion ::=
    subtype_mark(expression)
  | subtype_mark(name)

4.7:
qualified_expression ::=
   subtype_mark'(expression) | subtype_mark'aggregate
```

4.8:
allocator ::=
  new subtype_indication | **new** qualified_expression

5.1:
sequence_of_statements ::= statement {statement}

5.1:
statement ::=
  {label} simple_statement | {label} compound_statement

5.1:
```
simple_statement ::= null_statement
  | assignment_statement          | exit_statement
  | goto_statement                | procedure_call_statement
  | simple_return_statement       | entry_call_statement
  | requeue_statement             | delay_statement
  | abort_statement               | raise_statement
  | code_statement
```

5.1:
```
compound_statement ::=
    if_statement                  | case_statement
  | loop_statement                | block_statement
  | extended_return_statement
  | accept_statement              | select_statement
```

5.1:
null_statement ::= **null**;

5.1:
label ::= <<*label*_statement_identifier>>

5.1:
statement_identifier ::= direct_name

5.2:
assignment_statement ::=
  *variable*_name := expression;

5.3:
```
if_statement ::=
   if condition then
    sequence_of_statements
  {elsif condition then
    sequence_of_statements}
  [else
    sequence_of_statements]
   end if;
```

5.3:
condition ::= *boolean*_expression

5.4:
```
case_statement ::=
  case expression is
     case_statement_alternative
    {case_statement_alternative}
  end case;
```

5.4:
```
case_statement_alternative ::=
  when discrete_choice_list =>
    sequence_of_statements
```

5.5:
```
loop_statement ::=
  [loop_statement_identifier:]
    [iteration_scheme] loop
      sequence_of_statements
     end loop [loop_identifier];
```

5.5:
iteration_scheme ::= **while** condition
  | **for** loop_parameter_specification

5.5:
loop_parameter_specification ::=
  defining_identifier **in** [**reverse**] discrete_subtype_definition

5.6:
block_statement ::=
  [*block*_statement_identifier:]
    [**declare**
      declarative_part]
    begin
      handled_sequence_of_statements
    end [*block*_identifier];

5.7:
exit_statement ::=
  exit [*loop*_name] [**when** condition];

5.8:
goto_statement ::= **goto** *label*_name;

6.1:
subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification;

6.1:
subprogram_specification ::=
  procedure_specification
 | function_specification

6.1:
procedure_specification ::= **procedure** defining_program_unit_name parameter_profile

6.1:
function_specification ::= **function** defining_designator parameter_and_result_profile

6.1:
designator ::= [parent_unit_name .]identifier | operator_symbol

6.1:
defining_designator ::= defining_program_unit_name | defining_operator_symbol

6.1:
defining_program_unit_name ::= [parent_unit_name .]defining_identifier

6.1:
operator_symbol ::= string_literal

6.1:
defining_operator_symbol ::= operator_symbol

6.1:
parameter_profile ::= [formal_part]

6.1:
parameter_and_result_profile ::=
  [formal_part] **return** [null_exclusion] subtype_mark
 | [formal_part] **return** access_definition

6.1:
formal_part ::=
  (parameter_specification {; parameter_specification})

6.1:
parameter_specification ::=
  defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]

6.1:
mode ::= [**in**] | **in out** | **out**

6.3:
subprogram_body ::=
   [overriding_indicator]
   subprogram_specification **is**
      declarative_part
   begin
      handled_sequence_of_statements
   end [designator];

6.4:
procedure_call_statement ::=
   *procedure*_name;
 | *procedure*_prefix actual_parameter_part;

6.4:
function_call ::=
   *function*_name
 | *function*_prefix actual_parameter_part

6.4:
actual_parameter_part ::=
   (parameter_association {, parameter_association})

6.4:
parameter_association ::=
  [*formal_parameter*_selector_name =>] explicit_actual_parameter

6.4:
explicit_actual_parameter ::= expression | *variable*_name

6.5:
simple_return_statement ::= **return** [expression];

6.5:
extended_return_statement ::=
   return defining_identifier : [**aliased**] return_subtype_indication [:= expression] [**do**
      handled_sequence_of_statements
   end return];

6.5:
return_subtype_indication ::= subtype_indication | access_definition

6.7:
null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification **is null**;

7.1:
package_declaration ::= package_specification;

7.1:
package_specification ::=
   package defining_program_unit_name **is**
     {basic_declarative_item}
  [**private**
     {basic_declarative_item}]
   end [[parent_unit_name.]identifier]

7.2:
package_body ::=
   package body defining_program_unit_name **is**
      declarative_part
  [**begin**
      handled_sequence_of_statements]
   end [[parent_unit_name.]identifier];

7.3:
private_type_declaration ::=
  type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**;

7.3:
private_extension_declaration ::=
  type defining_identifier [discriminant_part] **is**
    [**abstract**] [**limited** | **synchronized**] **new** *ancestor*_subtype_indication
    [**and** interface_list] **with private**;

8.3.1:
overriding_indicator ::= [**not**] **overriding**

8.4:
use_clause ::= use_package_clause | use_type_clause

8.4:
use_package_clause ::= **use** *package*_name {, *package*_name};

8.4:
use_type_clause ::= **use type** subtype_mark {, subtype_mark};

8.5:
renaming_declaration ::=
      object_renaming_declaration
    | exception_renaming_declaration
    | package_renaming_declaration
    | subprogram_renaming_declaration
    | generic_renaming_declaration

8.5.1:
object_renaming_declaration ::=
   defining_identifier : [null_exclusion] subtype_mark **renames** *object*_name;
 | defining_identifier : access_definition **renames** *object*_name;

8.5.2:
exception_renaming_declaration ::= defining_identifier : **exception renames** *exception*_name;

8.5.3:
package_renaming_declaration ::= **package** defining_program_unit_name **renames** *package*_name;

8.5.4:
subprogram_renaming_declaration ::=
   [overriding_indicator]
   subprogram_specification **renames** *callable_entity*_name;

8.5.5:
generic_renaming_declaration ::=
   generic package   defining_program_unit_name **renames** *generic_package*_name;
 | **generic procedure**   defining_program_unit_name **renames** *generic_procedure*_name;
 | **generic function**   defining_program_unit_name **renames** *generic_function*_name;

9.1:
task_type_declaration ::=
  task type defining_identifier [known_discriminant_part] [**is**
    [**new** interface_list **with**]
    task_definition];

9.1:
single_task_declaration ::=
  task defining_identifier [**is**
    [**new** interface_list **with**]
    task_definition];

9.1:
task_definition ::=
    {task_item}
  [**private**
    {task_item}]
  end [*task*_identifier]

9.1:
task_item ::= entry_declaration | aspect_clause

9.1:
task_body ::=
  task body defining_identifier **is**
    declarative_part
  begin
    handled_sequence_of_statements
  end [*task*_identifier];

9.4:
protected_type_declaration ::=
  protected type defining_identifier [known_discriminant_part] **is**
    [**new** interface_list **with**]
    protected_definition;

9.4:
single_protected_declaration ::=
  protected defining_identifier **is**
    [**new** interface_list **with**]
    protected_definition;

9.4:
protected_definition ::=
    { protected_operation_declaration }
[**private**
    { protected_element_declaration }]
  end [*protected*_identifier]

9.4:
protected_operation_declaration ::= subprogram_declaration
    | entry_declaration
    | aspect_clause

9.4:
protected_element_declaration ::= protected_operation_declaration
    | component_declaration

9.4:
protected_body ::=
  protected body defining_identifier **is**
   { protected_operation_item }
  end [*protected*_identifier];

9.4:
protected_operation_item ::= subprogram_declaration
    | subprogram_body
    | entry_body
    | aspect_clause

9.5.2:
entry_declaration ::=
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile;

9.5.2:
accept_statement ::=
  accept *entry*_direct_name [(entry_index)] parameter_profile [**do**
    handled_sequence_of_statements
  end [*entry*_identifier]];

9.5.2:
entry_index ::= expression

9.5.2:
entry_body ::=
  entry defining_identifier entry_body_formal_part entry_barrier **is**
    declarative_part
  begin
    handled_sequence_of_statements
  end [*entry*_identifier];

9.5.2:
entry_body_formal_part ::= [(entry_index_specification)] parameter_profile

9.5.2:
entry_barrier ::= **when** condition

9.5.2:
entry_index_specification ::= **for** defining_identifier **in** discrete_subtype_definition

9.5.3:
entry_call_statement ::= *entry*_name [actual_parameter_part];

9.5.4:
requeue_statement ::= **requeue** *entry*_name [**with abort**];

9.6:
delay_statement ::= delay_until_statement | delay_relative_statement

9.6:
delay_until_statement ::= **delay until** *delay*_expression;

9.6:
delay_relative_statement ::= **delay** *delay*_expression;

9.7:
```
select_statement ::=
   selective_accept
  | timed_entry_call
  | conditional_entry_call
  | asynchronous_select
```

9.7.1:
```
selective_accept ::=
  select
   [guard]
     select_alternative
{ or
   [guard]
     select_alternative }
[ else
   sequence_of_statements ]
  end select;
```

9.7.1:
guard ::= **when** condition =>

9.7.1:
```
select_alternative ::=
   accept_alternative
  | delay_alternative
  | terminate_alternative
```

9.7.1:
```
accept_alternative ::=
  accept_statement [sequence_of_statements]
```

9.7.1:
```
delay_alternative ::=
  delay_statement [sequence_of_statements]
```

9.7.1:
terminate_alternative ::= **terminate**;

9.7.2:
```
timed_entry_call ::=
  select
   entry_call_alternative
  or
   delay_alternative
  end select;
```

9.7.2:
entry_call_alternative ::=
  procedure_or_entry_call [sequence_of_statements]

9.7.2:
procedure_or_entry_call ::=
  procedure_call_statement | entry_call_statement

9.7.3:
conditional_entry_call ::=
  select
   entry_call_alternative
  else
   sequence_of_statements
  end select;

9.7.4:
asynchronous_select ::=
  select
   triggering_alternative
  then abort
   abortable_part
  end select;

9.7.4:
triggering_alternative ::= triggering_statement [sequence_of_statements]

9.7.4:
triggering_statement ::= procedure_or_entry_call | delay_statement

9.7.4:
abortable_part ::= sequence_of_statements

9.8:
abort_statement ::= **abort** *task*_name {, *task*_name};

10.1.1:
compilation ::= {compilation_unit}

10.1.1:
compilation_unit ::=
    context_clause library_item
  | context_clause subunit

10.1.1:
library_item ::= [**private**] library_unit_declaration
  | library_unit_body
  | [**private**] library_unit_renaming_declaration

10.1.1:
library_unit_declaration ::=
     subprogram_declaration | package_declaration
   | generic_declaration | generic_instantiation

10.1.1:
library_unit_renaming_declaration ::=
   package_renaming_declaration
 | generic_renaming_declaration
 | subprogram_renaming_declaration

10.1.1:
library_unit_body ::= subprogram_body | package_body

10.1.1:
parent_unit_name ::= name

10.1.2:
context_clause ::= {context_item}

10.1.2:
context_item ::= with_clause | use_clause

10.1.2:
with_clause ::= limited_with_clause | nonlimited_with_clause

10.1.2:
limited_with_clause ::= **limited** [**private**] **with** *library_unit_*name {, *library_unit_*name};

10.1.2:
nonlimited_with_clause ::= [**private**] **with** *library_unit_*name {, *library_unit_*name};

10.1.3:
body_stub ::= subprogram_body_stub | package_body_stub | task_body_stub | protected_body_stub

10.1.3:
subprogram_body_stub ::=
  [overriding_indicator]
  subprogram_specification **is separate**;

10.1.3:
package_body_stub ::= **package body** defining_identifier **is separate**;

10.1.3:
task_body_stub ::= **task body** defining_identifier **is separate**;

10.1.3:
protected_body_stub ::= **protected body** defining_identifier **is separate**;

10.1.3:
subunit ::= **separate** (parent_unit_name) proper_body

11.1:
exception_declaration ::= defining_identifier_list : **exception**;

11.2:
handled_sequence_of_statements ::=
    sequence_of_statements
  [**exception**
    exception_handler
   {exception_handler}]

11.2:
exception_handler ::=
  when [choice_parameter_specification:] exception_choice {| exception_choice} =>
    sequence_of_statements

11.2:
choice_parameter_specification ::= defining_identifier

11.2:
exception_choice ::= *exception_*name | **others**

11.3:
raise_statement ::= **raise**;
     | **raise** *exception_*name [**with** *string_*expression];

12.1:
generic_declaration ::= generic_subprogram_declaration | generic_package_declaration

12.1:
generic_subprogram_declaration ::=
    generic_formal_part subprogram_specification;

12.1:
generic_package_declaration ::=
    generic_formal_part package_specification;

12.1:
generic_formal_part ::= **generic** {generic_formal_parameter_declaration | use_clause}

12.1:
generic_formal_parameter_declaration ::=
     formal_object_declaration
   | formal_type_declaration
   | formal_subprogram_declaration
   | formal_package_declaration

12.3:
generic_instantiation ::=
    package defining_program_unit_name **is**
        new *generic_package*_name [generic_actual_part];
  | [overriding_indicator]
    procedure defining_program_unit_name **is**
        new *generic_procedure*_name [generic_actual_part];
  | [overriding_indicator]
    function defining_designator **is**
        new *generic_function*_name [generic_actual_part];

12.3:
generic_actual_part ::=
  (generic_association {, generic_association})

12.3:
generic_association ::=
  [*generic_formal_parameter*_selector_name =>] explicit_generic_actual_parameter

12.3:
explicit_generic_actual_parameter ::= expression | *variable*_name
  | *subprogram*_name | *entry*_name | subtype_mark
  | *package_instance*_name

12.4:
formal_object_declaration ::=
   defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression];
   defining_identifier_list : mode access_definition [:= default_expression];

12.5:
formal_type_declaration ::=
   type defining_identifier[discriminant_part] **is** formal_type_definition;

12.5:
formal_type_definition ::=
     formal_private_type_definition
   | formal_derived_type_definition
   | formal_discrete_type_definition
   | formal_signed_integer_type_definition
   | formal_modular_type_definition
   | formal_floating_point_definition
   | formal_ordinary_fixed_point_definition
   | formal_decimal_fixed_point_definition
   | formal_array_type_definition
   | formal_access_type_definition
   | formal_interface_type_definition

12.5.1:
formal_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] **private**

12.5.1:
formal_derived_type_definition ::=
    [**abstract**] [**limited** | **synchronized**] **new** subtype_mark [[**and** interface_list]**with private**]

12.5.2:
formal_discrete_type_definition ::= (<>)

12.5.2:
formal_signed_integer_type_definition ::= **range** <>

12.5.2:
formal_modular_type_definition ::= **mod** <>

12.5.2:
formal_floating_point_definition ::= **digits** <>

12.5.2:
formal_ordinary_fixed_point_definition ::= **delta** <>

12.5.2:
formal_decimal_fixed_point_definition ::= **delta** <> **digits** <>

12.5.3:
formal_array_type_definition ::= array_type_definition

12.5.4:
formal_access_type_definition ::= access_type_definition

12.5.5:
formal_interface_type_definition ::= interface_type_definition

12.6:
formal_subprogram_declaration ::= formal_concrete_subprogram_declaration
   | formal_abstract_subprogram_declaration

12.6:
formal_concrete_subprogram_declaration ::=
   with subprogram_specification [**is** subprogram_default];

12.6:
formal_abstract_subprogram_declaration ::=
   with subprogram_specification **is abstract** [subprogram_default];

12.6:
subprogram_default ::= default_name | <> | **null**

12.6:
default_name ::= name

12.7:
formal_package_declaration ::=
   with package defining_identifier **is new** *generic_package_*name formal_package_actual_part;

12.7:
formal_package_actual_part ::=
   ([**others** =>] <>)
 | [generic_actual_part]
 | (formal_package_association {, formal_package_association} [, **others** => <>])

12.7:
formal_package_association ::=
   generic_association
 | *generic_formal_parameter_*selector_name => <>

13.1:
aspect_clause ::= attribute_definition_clause
   | enumeration_representation_clause
   | record_representation_clause
   | at_clause

13.1:
local_name ::= direct_name
   | direct_name'attribute_designator
   | *library_unit_*name

13.3:
attribute_definition_clause ::=
   for local_name'attribute_designator **use** expression;
 | **for** local_name'attribute_designator **use** name;

13.4:
enumeration_representation_clause ::=
   for *first_subtype_*local_name **use** enumeration_aggregate;

13.4:
enumeration_aggregate ::= array_aggregate

13.5.1:
record_representation_clause ::=
   for *first_subtype_*local_name **use**
    record [mod_clause]
     {component_clause}
    end record;

13.5.1:
component_clause ::=
  *component*_local_name **at** position **range** first_bit .. last_bit;

13.5.1:
position ::= *static*_expression

13.5.1:
first_bit ::= *static*_simple_expression

13.5.1:
last_bit ::= *static*_simple_expression

13.8:
code_statement ::= qualified_expression;

13.12:
restriction ::= *restriction*_identifier
  | *restriction_parameter*_identifier => restriction_parameter_argument

13.12:
restriction_parameter_argument ::= name | expression

J.3:
delta_constraint ::= **delta** *static*_expression [range_constraint]

J.7:
at_clause ::= **for** direct_name **use at** expression;

J.8:
mod_clause ::= **at mod** *static*_expression;

Syntax Cross Reference

{*syntax (cross reference)*} {*grammar (cross reference)*} {*context free grammar (cross reference)*} {*BNF (Backus-Naur Form) (cross reference)*} {*Backus-Naur Form (BNF) (cross reference)*}

In the following syntax cross reference, each syntactic category is followed by the clause number where it is defined. In addition, each syntactic category *S* is followed by a list of the categories that use *S* in their definitions. For example, the first listing below shows that abort_statement appears in the definition of simple_statement. 1

abort_statement		9.8
	simple_statement	5.1
abortable_part		9.7.4
	asynchronous_select	9.7.4
abstract_subprogram_declaration		3.9.3
	basic_declaration	3.1
accept_alternative		9.7.1
	select_alternative	9.7.1
accept_statement		9.5.2
	accept_alternative	9.7.1
	compound_statement	5.1
access_definition		3.10
	component_definition	3.6
	discriminant_specification	3.7
	formal_object_declaration	12.4
	object_declaration	3.3.1
	object_renaming_declaration	8.5.1
	parameter_and_result_profile	6.1
	parameter_specification	6.1
	return_subtype_indication	6.5
access_to_object_definition		3.10
	access_type_definition	3.10
access_to_subprogram_definition		3.10
	access_type_definition	3.10
access_type_definition		3.10
	formal_access_type_definition	12.5.4
	type_definition	3.2.1
actual_parameter_part		6.4
	entry_call_statement	9.5.3
	function_call	6.4
	procedure_call_statement	6.4
aggregate		4.3
	primary	4.4
	qualified_expression	4.7
allocator		4.8
	primary	4.4
ancestor_part		4.3.2
	extension_aggregate	4.3.2
array_aggregate		4.3.3
	aggregate	4.3
	enumeration_aggregate	13.4
array_component_association		4.3.3
	named_array_aggregate	4.3.3
array_type_definition		3.6
	formal_array_type_definition	12.5.3
	object_declaration	3.3.1
	type_definition	3.2.1
aspect_clause		13.1
	basic_declarative_item	3.11
	component_item	3.8
	protected_operation_declaration	9.4
	protected_operation_item	9.4
	task_item	9.1
assignment_statement		5.2
	simple_statement	5.1
asynchronous_select		9.7.4
	select_statement	9.7
at_clause		J.7
	aspect_clause	13.1
attribute_definition_clause		13.3
	aspect_clause	13.1
attribute_designator		4.1.4
	attribute_definition_clause	13.3
	attribute_reference	4.1.4
	local_name	13.1
attribute_reference		4.1.4
	name	4.1
base		2.4.2
	based_literal	2.4.2
based_literal		2.4.2
	numeric_literal	2.4
based_numeral		2.4.2
	based_literal	2.4.2
basic_declaration		3.1
	basic_declarative_item	3.11
basic_declarative_item		3.11
	declarative_item	3.11
	package_specification	7.1
binary_adding_operator		4.5
	simple_expression	4.4
block_statement		5.6
	compound_statement	5.1
body		3.11
	declarative_item	3.11

body_stub		10.1.3
	body	3.11
case_statement		5.4
	compound_statement	5.1
case_statement_alternative		5.4
	case_statement	5.4
character		2.1
	comment	2.7
character_literal		2.5
	defining_character_literal	3.5.1
	name	4.1
	selector_name	4.1.3
choice_parameter_specification		11.2
	exception_handler	11.2
code_statement		13.8
	simple_statement	5.1
compilation_unit		10.1.1
	compilation	10.1.1
component_choice_list		4.3.1
	record_component_association	4.3.1
component_clause		13.5.1
	record_representation_clause	13.5.1
component_declaration		3.8
	component_item	3.8
	protected_element_declaration	9.4
component_definition		3.6
	component_declaration	3.8
	constrained_array_definition	3.6
	unconstrained_array_definition	3.6
component_item		3.8
	component_list	3.8
component_list		3.8
	record_definition	3.8
	variant	3.8.1
composite_constraint		3.2.2
	constraint	3.2.2
compound_statement		5.1
	statement	5.1
condition		5.3
	entry_barrier	9.5.2
	exit_statement	5.7
	guard	9.7.1
	if_statement	5.3
	iteration_scheme	5.5
conditional_entry_call		9.7.3
	select_statement	9.7
constrained_array_definition		3.6
	array_type_definition	3.6
constraint		3.2.2
	subtype_indication	3.2.2
context_clause		10.1.2
	compilation_unit	10.1.1
context_item		10.1.2
	context_clause	10.1.2
decimal_fixed_point_definition		3.5.9
	fixed_point_definition	3.5.9
decimal_literal		2.4.1
	numeric_literal	2.4
declarative_item		3.11
	declarative_part	3.11
declarative_part		3.11
	block_statement	5.6
	entry_body	9.5.2
	package_body	7.2
	subprogram_body	6.3
	task_body	9.1
default_expression		3.7
	component_declaration	3.8
	discriminant_specification	3.7
	formal_object_declaration	12.4
	parameter_specification	6.1
default_name		12.6
	subprogram_default	12.6
defining_character_literal		3.5.1
	enumeration_literal_specification	3.5.1
defining_designator		6.1
	function_specification	6.1
	generic_instantiation	12.3
defining_identifier		3.1
	choice_parameter_specification	11.2
	defining_identifier_list	3.3.1
	defining_program_unit_name	6.1
	entry_body	9.5.2
	entry_declaration	9.5.2
	entry_index_specification	9.5.2
	enumeration_literal_specification	3.5.1
	exception_renaming_declaration	8.5.2
	extended_return_statement	6.5
	formal_package_declaration	12.7
	formal_type_declaration	12.5
	full_type_declaration	3.2.1
	incomplete_type_declaration	3.10.1
	loop_parameter_specification	5.5
	object_renaming_declaration	8.5.1
	package_body_stub	10.1.3
	private_extension_declaration	7.3
	private_type_declaration	7.3
	protected_body	9.4
	protected_body_stub	10.1.3
	protected_type_declaration	9.4
	single_protected_declaration	9.4
	single_task_declaration	9.1
	subtype_declaration	3.2.2
	task_body	9.1
	task_body_stub	10.1.3
	task_type_declaration	9.1

Annex Q
(informative)
Language-Defined Entities

{*AI95-00440-01*} This annex lists the language-defined entities of the language. A list of language-defined library units can be found in Annex A, “Predefined Language Environment”. 1/2

Q.1 Language-Defined Packages

{*AI95-00440-01*} This clause lists all language-defined packages. 1/2

Ada   A.2(2)
Address_To_Access_Conversions
  child of System   13.7.2(2)
Arithmetic
  child of Ada.Calendar   9.6.1(8/2)
ASCII
  in Standard   A.1(36.3/2)
Assertions
  child of Ada   11.4.2(12/2)
Asynchronous_Task_Control
  child of Ada   D.11(3/2)
Bounded
  child of Ada.Strings   A.4.4(3)
Bounded_IO
  child of Ada.Text_IO   A.10.11(3/2)
  child of Ada.Wide_Text_IO   A.11(4/2)
  child of Ada.Wide_Wide_Text_IO   A.11(4/2)
C
  child of Interfaces   B.3(4)
Calendar
  child of Ada   9.6(10)
Characters
  child of Ada   A.3.1(2)
COBOL
  child of Interfaces   B.4(7)
Command_Line
  child of Ada   A.15(3)
Complex_Arrays
  child of Ada.Numerics   G.3.2(53/2)
Complex_Elementary_Functions
  child of Ada.Numerics   G.1.2(9/1)
Complex_Text_IO
  child of Ada   G.1.3(9.1/2)
Complex_Types
  child of Ada.Numerics   G.1.1(25/1)
Complex_IO
  child of Ada.Text_IO   G.1.3(3)
  child of Ada.Wide_Text_IO   G.1.4(1)
  child of Ada.Wide_Wide_Text_IO   G.1.5(1/2)
Constants
  child of Ada.Strings.Maps   A.4.6(3/2)
Containers
  child of Ada   A.18.1(3/2)
Conversions
  child of Ada.Characters   A.3.4(2/2)
Decimal
  child of Ada   F.2(2)
Decimal_Conversions
  in Interfaces.COBOL   B.4(31)
Decimal_IO
  in Ada.Text_IO   A.10.1(73)
Decimal_Output
  in Ada.Text_IO.Editing   F.3.3(11)
Direct_IO
  child of Ada   A.8.4(2)
Directories
  child of Ada   A.16(3/2)
Discrete_Random
  child of Ada.Numerics   A.5.2(17)
Dispatching
  child of Ada   D.2.1(1.2/2)
Doubly_Linked_Lists
  child of Ada.Containers   A.18.3(5/2)
Dynamic_Priorities
  child of Ada   D.5.1(3/2)
EDF
  child of Ada.Dispatching   D.2.6(9/2)
Editing
  child of Ada.Text_IO   F.3.3(3)
  child of Ada.Wide_Text_IO   F.3.4(1)
  child of Ada.Wide_Wide_Text_IO   F.3.5(1/2)
Elementary_Functions
  child of Ada.Numerics   A.5.1(9/1)
Enumeration_IO
  in Ada.Text_IO   A.10.1(79)
Environment_Variables
  child of Ada   A.17(3/2)
Exceptions
  child of Ada   11.4.1(2/2)
Execution_Time
  child of Ada   D.14(3/2)
Finalization
  child of Ada   7.6(4/1)
Fixed
  child of Ada.Strings   A.4.3(5)
Fixed_IO
  in Ada.Text_IO   A.10.1(68)
Float_Random
  child of Ada.Numerics   A.5.2(5)
Float_Text_IO
  child of Ada   A.10.9(33)

Q.2 Language-Defined Types and Subtypes

{*AI95-00440-01*} This clause lists all language-defined types and subtypes. 1/2

Q.3 Language-Defined Subprograms

1/2 {*AI95-00440-01*} This clause lists all language-defined subprograms.

Q.4 Language-Defined Exceptions

1/2 {*AI95-00440-01*} This clause lists all language-defined exceptions.

Q.5 Language-Defined Objects

{*AI95-00440-01*} This clause lists all language-defined constants, variables, named numbers, and enumeration literals. 1/2

To be honest: Formally, named numbers and enumeration literals aren't objects, but it was thought to be too weird to say “Language-Defined Objects and Values”. 1.a/2

Index

Index entries are given by paragraph number.