AVA 95 Reference Manual

Language and Standard Libraries

Modifications by
Michael K. Smith and Robert L. Akers

5 October 1995

Derived from ISO/IEC JTC1/SC22 WG9 N 193, AARM Version 6.0

CLI Technical Report 114

Computational Logic, Inc.
1717 W. 6th, Suite 290
Austin, Texas 78703
(512) 322-9951

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION ELECTROTECHNICAL COMMISSION

Original text published by
Intermetrics, Inc.
733 Concord Avenue
Cambridge, Massachusetts 02138

Modified for AVA by
Computational Logic, Inc.
1717 W. 6th, Suite 290
Austin, Texas 78703

Foreword

AVA modifications were supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

Contents

Forward to the AVA Revision

1 AVA (A Verifiable Ada) is an attempt to *formally* define a subset of the Ada programming language sufficient for reasonably sized programming projects. Such a formal definition is a prerequisite to the production of provably correct Ada programs. This document in general is a subset of [ISO 94] and represents the *informal* description of AVA. The formal dynamic semantic definition is described in [Smith 95].

2 We have removed or constrained various language elements. Not all of these changes were motivated by the needs of formal definition. Some constructs were removed just to simplify this effort. Certain constructs, while amenable to formal definition, were removed because it was not clear how such a formalization would be used to prove properties about programs.

3 We have indicated those places where we have deleted or re-worded text. Large blocks of text (like chapters and sections) that have been deleted are indicated by ‘‘removed’’. Sections and subsections that have been added are marked with ‘‘new’’. Paragraphs, sentences, and portions thereof that have been removed are indicated by a ‘‘♦’’. The deletion of a series of paragraphs can be detected by observing the discontinuity in paragraph numbers. In some places we have modified or added text for clarification or to state stronger restrictions than Ada. This text appears in facecode Helvetica. Changes to syntactic category names, which in the Ada manual are sans-serif, e.g. parameter_association, are indicated by bold sans-serif, **inner_declaration**. The Ada Manual uses roman italics to indicate semantic constraints on syntactic categories, e.g. *procedure_*name. If we change these, we use Helvetica italics, e.g *function_*name. Deletions in Appendices other than Appendix A have *not* been scrupulously tracked.

4 This document is based on the on-line version of *Programming Language Ada, Language and Standard Libraries, Annotated Version 6.0* [ISO 94] available at ajpo.sei.cmu.com as well as the online ascii version, *Ada 95 Reference Manual* available through
http://lglwww.eplf.ch/Ada/LRM/9X/RM/Text/aarm.doc
(hereafter AARM). Our thanks to Tucker Taft and Intermetrics for making available the Scribe input for draft version 5.0 of the manual, as well as the assorted macros that handle paragraph numbering and appendix creation.

5 This modified version was created by Michael K. Smith and Robert L. Akers of Computational Logic, Inc. Substantial discussions on the details of restrictions as they applied to the language described in the original Ada Reference Manual language [DoD 83] (hereafter ARM83) were carried out with Dan Craigen and Mark Saaltink (now of Odyssey Research Associates). Many of the detailed modifications were inspired by the extensive discussions available in the accumulated Ada Interpretations.

6 **Predictability and critical systems**

7 Computational Logic, Inc. is concerned with the ultimate goal of fielding *highly predictable* systems. Eventually we expect that all of the links in the chain of system development, from high level language to hardware, will be amenable to predictability analysis. (See for example the December 89 issue of *Journal of Automated Reasoning* which contains four articles describing the ‘‘Computational Logic Short Stack’’.) One of the requirements for predicting the behavior of a program written in a high level language is a precise understanding of the expected behavior of language constructs. This manual represents an effort to carve out a predictable subset of the Ada programming language.

8 Applications with a requirement for *high predictability* include security oriented and safety critical systems. Real-time applications have a significant need for detailed predictability in order to assess the capability of the application to meet hard timing deadlines. Eventually we would hope that predictability would be a requirement of *all* Ada programs.

9 Other work

10 There have been two motivations for work on Ada subsets.

11 1. To define a dialect with predictable behavior for safety and security critical systems.

12 2. To carve out a subset for which a reasonably tractable formal definition can be provided.

13 The second is ultimately in support of the first.

14 In addition there have been efforts to provide a *complete* formal definition of Ada [DDC 87] in conformance with the published standard [DoD 83].

15 A SETL interpreter for Ada was developed at NYU [Courant 84, Courant 83]. However, it appears that the requirement of reasonable efficiency makes the definition more opaque than we would like a formal definition to be.

16 The Ada Runtime Environment Working Group (ARTEWG) produced a *Catalogue of Ada Runtime Implementation Dependencies* [ARTEWG 87].

> The main goal of this catalogue is to be the one place where all the areas of the Ada Reference Manual (RM) which permit implementation flexibilities can be found.

This effort was primarily in aid of predictability and portability.

17 The European Economic Community supported an attempt to provide a *complete* formal definition of Ada [DDC 87] in conformance with the published standard, the *Reference Manual for the Ada Programming Language* [DoD 83] (ARM83). Conformance to the complete ARM83 presents some unsolvable problems. The EEC definition was unable to define parts of the language because the definition embodied in the ARM83 is ambiguous. It does a great service by detailing these problems. One drawback to the EEC definition is its size. The definition is contained in 8 loose leaf binders and depends on several supporting documents.

18 We have two observations with regard to the EEC definition.

19 • It clearly indicates that a formal definition of a programming language as complex as Ada is possible. If the research team had been able to depart from the ARM83 and make some minor modifications, they would have been able to complete their definition.

20 • Building tools to support formal reasoning from a definition this complex is problematic. We believe that any successful tool of this sort will need to be based on a simpler formal description, presumably for a subset of the language.

21 ORA has produced the "Penelope System" [Ramsey 88, Polak 88] which has been used to prove properties of some significant Ada programs. It is based on a formal definition for a language that corresponds to a substantial subset of Ada 83. This language has a more regular semantics than a literal Ada definition would.

22 Carre has developed a subset, SPARK (SPADE Ada Kernel) [Carre 88], which is an ‘‘annotated sublanguage of Ada, intended for use in safety-critical applications’’. It is supported by tools in the SPADE

system, available from PRAXIS PVL. Of recent note is the publication of a formal semantics for the SPARK subset written in the Z notation [Marsh 94, O'Neill 94].

Foreword to the Original Annotated Ada Reference Manual

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 This document is an annotated subset of the second edition which canceled and replaced the first edition (ISO 8652:1987), of which it constituted a technical revision.

Introduction

1 This is version 1.0 of the AVA 95 Reference Manual (AVARM). Comments on this document are welcome; see the Instructions for Comment Submission below.

2 Other available Ada documents include:

3 • Rationale for the Ada Programming Language -- 1995 edition, which gives an introduction to the new features of Ada, and explains the rationale behind them. ♦

4 • The Ada Reference Manual (RM). This is the International Standard — ISO/IEC 8652:1995(E).

5 • The Annotated Ada Reference Manual (AARM). The AARM [ISO 94] contains all of the text in the RM, plus various annotations. It is intended primarily for compiler writers, validation test writers, and others who wish to study the fine details. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

6 • Changes to Ada -- 1987 to 1995. This document lists in detail the changes made to the 1987 edition of the standard.

Design Goals -- Removed

Language Summary

11 An AVA program is composed of one or more program units. Program units may be subprograms (which define executable algorithms) or packages (which define collections of entities) ♦. 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 AVA 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 ♦

19 ♦

20 *Declarations and Statements*

21 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.

22 The declarative part associates names with declared entities. For example, a name may denote a subtype, a constant, or a variable ♦. A declarative part also introduces the names and parameters of ♦ nested subprograms or packages♦ to be used in the program unit.

23 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).

24 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.

25 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.

26 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.

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

28 ♦

29 Certain declarations and statements in AVA are *assertions*. These establish logical requirements for various program states. These assertions are statements in the ACL2 logic with respect to program states. The formal definition provides the means to link these assertions to program behavior. In addition to assertions, AVA provides a means to define axioms, conjectures, and logical functions in the ACL2 logic for use in program proof.

30 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.

31 *Data Types*

32 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 and numeric♦) and composite types (including array and record types).

33 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 and Character♦ are predefined.

34 Numeric types provide a means of performing exact ♦ numerical computations. Exact computations use integer types, which denote sets of consecutive integers. ♦ The numeric type Integer♦ is predefined.

35 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. ♦ The array type String ♦ is predefined.

36 ♦

37 ♦

38 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.

39 ♦

40 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 and arrays with a limited set of index values♦.

41 *Other Facilities*

42 ♦

43 The predefined environment of the language provides for input-output and other capabilities (such as string manipulation ♦) by means of standard library packages. Input-output is supported for values of ♦ Character and String types. ♦

44 ♦

Language Changes

♦

Acknowledgements (from the Original Annotated Ada Reference Manual)

67 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.

68 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).

69 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).

70 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.

71 Special thanks go to R. Mathis, Convenor of ISO/IEC JTC1/SC22 Working Group 9.

72 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.

Instructions for AVA Comment Submission

73 Comments should be sent via one of the following methods:

74

US Mail:	Micheal K. Smith Computational Logic, Inc. 1717 W. 6th, Suite 290 Austin, Texas 78703
Phone:	(512) 322-9951
FAX:	(512) 322-0565 Attn: Micheal K. Smith
E-Mail:	**mksmith@cli.com**

75 Please use e-mail if at all possible.

76 Comments should use the following format:

77

!topic *Title summarizing comment on AVARM*
!reference AVARM-*ss.ss(pp)*;1.0
!from *Author Name yy-mm-dd*
!keywords *keywords related to topic*
!discussion

78 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. References to multiple sections, clauses and/or subclauses can be made by including additional **!reference** lines in the comment. As noted above the version of this Reference Manual is 1.0.

79 ♦

80 Multiple related comments per e-mail message are acceptable, but in any case be sure that your e-mail ‘‘Subject’’ line says something specific like ‘‘Private Type Issues’’ rather than something general like ‘‘Comment on AVARM.’’

81 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:

82

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

83 Thank you for your help.

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 operations may be implemented as subprograms using conventional sequential control structures♦. The language treats modularity in the physical sense as well, with a facility to support separate compilation. AVA (A Verifiable Ada) is a subset of Ada. The purpose of AVA is to promote specification and proofs of properties of programs written within this subset. 1

♦ In AVA, errors can be signaled as exceptions and handled explicitly. ♦ Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output♦. 2

1.1 Scope

AVA95 (A Verifiable Ada) is a subset of Ada95. This Reference Manual specifies the form and meaning of programs written in AVA95. The purpose of the AVA subset is to promote formal specification and proofs of properties of programs written within this subset. 1

1.1.1 Extent of the Standard

This Reference Manual specifies: 1

- The form of a program written in AVA; 2
- The effect of translating and executing such a program; 3
- The manner in which program units may be combined to form AVA programs; 4
- The language-defined library units that a conforming implementation is required to supply; 5
- The permissible variations within the standard, and the manner in which they are to be documented; 6
- 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; 7
- ♦ 8
- The form of logical annotations and assertions and the interpretation of such annotations. 9

This Reference Manual 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
- ♦ 15
- The size of a program or program unit that will exceed the capacity of a particular conforming implementation. 16

17 In some places this standard requires more specific behavior from a conforming implementation than Ada does. For example, AVA specifies that on return from a procedure call the values of all **out** parameters are converted to the subtype of their respective actuals and only then copied back. Such places are marked with **AVA Implementation Requirement**. See section 1.1.5.

16 We exclude some constructs that Ada admits, for example Wide_character. As a result, there are some legal AVA programs that would not be legal Ada programs, for example due to use of a name defined in package Standard that we have deleted. This minor inconsistency could be corrected by preprocessing AVA programs. More difficult are type-related problems. For example, by 3.5.2 (9.a):

> The presence of Wide_Character in package Standard means that an expression such as
>
> ```
> 'a' = 'b'
> ```
>
> is ambiguous in Ada 95, whereas in Ada 83 both literals could be resolved to be of type Character.

The same lack of ambiguity exists in AVA.

1.1.2 Structure

1 This Reference Manual contains various sections and annexes, numbered to conform to the corresponding sections of AARM95, and an index.

2 The *core* of the AVA language consists of:

3 • Sections 1 through 13

4 • Annex A, ‘‘Predefined Language Environment’’

5 • ♦

6 • ♦

7 The following *Specialized Needs Annexes* define features that are needed by certain application areas:

8 • ♦

9 • ♦

10 • ♦

11 • ♦

12 • ♦

13 • Annex H, ‘‘Safety and Security’’

14

15 The core language and the Specialized Needs Annexes are normative, except that the material in each of the items listed below is informative:

16 • Text under a NOTES or Examples heading.

17 • Each clause or subclause whose title starts with the word ‘‘Example’’ or ‘‘Examples’’.

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

The following Annexes are informative: 19

- Annex J, ‘‘Language-Defined Attributes’’ 20
- ♦ 21
- Annex L, ‘‘Implementation-Defined Characteristics’’ 22
- Annex M, ‘‘Glossary’’ 23
- Annex N, ‘‘Syntax Summary’’ 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: ♦ 25

Syntax

Syntax rules (indented). 26

Name Resolution Rules

Compile-time rules that are used in name resolution, including overload resolution. 27

Formal Name Resolution Rules

A formalization of the compile-time rules used in name resolution, including overload resolution. 28

Legality Rules

Rules that are enforced at compile time. A construct is *legal* if it obeys all of the Legality Rules. 27

Static Semantics

A definition of the compile-time effect of each construct. 28

Abstract Syntax

The abstract syntax used to represent instances of the construct in the formal static and dynamic semantics. Before overload resolution some of these will be ambiguous and will be so marked. For example: 29

apply ∈ Apply	== **apply** *expr apl*	Before overload resolution.
function-call ∈ FunctionCall	== **function-call** *uid expr**	After overload resolution.

Thus, an occurence of *function-call* is of type FunctionCall, with a structure of the form **function-call** *uid expr**. We use * to indicate 0 or more occurences of a component.

Formal Static Semantics

A formal definition of the compile-time effect of each construct. 30

Post-Compilation Rules

Rules that are enforced before running a partition. A partition is legal if its compilation units are legal and it obeys all of the Post-Compilation Rules. 29

Dynamic Semantics

A definition of the run-time effect of each construct. 30

Formal Dynamic Semantics

31 A formal definition of the run-time effect of each construct.

Bounded (Run-Time) Errors

31 Situations that result in bounded (run-time) errors (see 1.1.5).

Erroneous Execution

32 We do not allow situations that result in erroneous execution (see 1.1.5).

Implementation Requirements

33 Additional requirements for conforming implementations.

Documentation Requirements

34 Documentation requirements for conforming implementations.

♦

NOTES

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

Examples

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

1.1.3 Conformity of an Implementation with the Standard

Implementation Requirements

1 A conforming implementation shall:

2 • Translate and correctly execute legal programs written in AVA, provided that they are not so large as to exceed the capacity of the implementation;

3 • 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);

4 • Identify all programs or program units that contain errors whose detection is required by this Reference Manual;

5 • Supply all language-defined library units required by this Reference Manual;

6 • ♦

7 • Specify all such variations in the manner prescribed by this Reference Manual.

8 The *external effect* of the execution of an AVA program is defined in terms of its interactions with its external environment. The following are defined as *external interactions*:

9 • Any interaction with an external file (see A.7);

10 • ♦

11 • ♦

12 • Any result returned or exception propagated from a main subprogram (see 10.2) ♦ to an external caller;

13 • ♦

14 • ♦

A conforming implementation of this Reference Manual shall produce for the execution of a given AVA program a set of interactions with the external environment whose order ♦ is consistent with the definitions and requirements of this Reference Manual for the semantics of the given program. 15

An implementation that conforms to this Standard shall support each capability required by the core language as specified. ♦ 16

In some places the AVA standard requires more specific behavior from a conforming implementation than Ada does. For example, AVA specifies that all actual parameters be passed by value in a subprogram call. Such places are marked with **AVA Implementation Requirement**. See also Section 1.1.5. 16

An implementation conforming to this Reference Manual may provide additional attributes and library units♦. The specification of these units must adhere to the AVA subset. However, it shall not provide any attribute or library unit ♦ having the same name as an attribute or library unit ♦ (respectively) specified in a Specialized Needs Annex of ARM95♦. 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

♦

Implementation Advice

If an implementation detects the use of an unsupported Ada95 Specialized Needs Annex feature at run time, it should raise Program_Error ♦. 20

♦ 21

1.1.4 Method of Description and Syntax Notation

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

The informal meaning of AVA programs is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs. The *formal* meaning of AVA programs is described by means of logical forms defining both the effects of each construct and the composition rules for constructs. 2

The context-free syntax of the language is described using a simple variant of Backus-Naur Form. In particular: 3

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

  case_statement 5

  AVA modifications to these catagories are in bold sans-serif, for example:

  inner_declaration 6

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

  array 7

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

9 return_statement ::= **return** [expression];
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 • 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 ♦ is equivalent to name alone. AVA modifications are indicated by bold italics, for example ***subtype_*****name**.

15 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 A *construct* is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under ‘‘Syntax.’’ ♦

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

18 ♦

NOTES

19 2 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 3 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.

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 AVA implementation;

3 These errors correspond to any violation of a rule given in this Reference Manual, 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 AVA 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.

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

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

The corresponding error situations are associated with the names of the predefined exceptions. Every AVA 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 be raised in every execution of a construct, then an implementation is allowed (although not required) to report this fact at compilation time. 6

- Bounded errors; 7

The Ada95 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. The errors of this category are called *bounded errors*. In Ada95, 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. AVA95 has selected **one** of the possible effects as the required behavior. 8

- ♦ 9

Note the elimination of erroneous execution and bounded errors. This has been accomplished in two ways. 10

1. We have removed or restricted some constructs that permit the kinds of ambiguity that lead to erroneous behavior. 10
2. We have specified *one* of the allowed Ada behaviors to be the allowed AVA behavior. These are marked with **AVA Implementation Requirement** in the text. 10

Order of evaluation in all important cases (e.g. state changing cases) is now specified. This makes the semantics much simpler, even though it assigns meanings to erroneous programs. It is our contention that virtually all substantial Ada applications that handle predefined exceptions are erroneous, so we do not feel that this represents any loss. For purposes of formal reasoning, it is certainly preferable to Ada's stance that the behavior of such programs is *a priori* unpredictable. In addition, there exists a simple preprocessing step to guarantee consistency with the AVA definition under any conforming Ada compiler.[1] This involves an Ada to Ada transformation that serializes those operations that have an undefined order in Ada so that their order of elaboration/evaluation corresponds to that prescribed for AVA. In addition we require value-result semantics for procedure calls. Again, this can be guaranteed by wrapping assignments to temporary variables around procedure calls. In the text we label certain progamming practices as resulting in programs whose ‘‘behavior will be dificult to predict’’. In general, these correspond to practices that can lead to compiler dependent behavior, even when executing code compiled by AVA conformant Ada compilers. 10

[1]Such transformations do require assumptions about the extent to which the compiler will optimize. An aggressive, optimizing compiler that does not ensure the visible behavioral equivalence between the original code and the optimized object is dangerous and unpredictable.

Implementation Permissions

11 An AVA 95 implementation may provide *nonstandard modes* of operation. Typically these modes would be selected ♦ 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♦. In any case, an implementation shall support a *standard* mode that conforms to the requirements of this Reference Manual; in particular, in the standard mode, all legal compilation_units shall be accepted.

♦

1.2 Normative References

1 The following standards may contain provisions which, through reference in this text, constitute provisions of this Reference Manual. At the time of original publication July, 1995, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this Reference Manual 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 *Information Processing — 7-bit Single-Byte Coded Character Set*

3 ♦

4 ♦

5 ISO/IEC 6429:1992 *Information Technology — Control Functions for Coded Character Sets*

6 ISO/IEC 8859-1:1987 *Information Processing — 8-bit Single-Byte Coded Character Sets — Part 1: Latin Alphabet No. 1*

7 ♦

8 ISO/IEC 10646-1:1993 *Information Technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

9 ♦

1.3 Definitions

1 Terms are defined throughout this Reference Manual, indicated by *italic* type. Terms explicitly defined in this Reference Manual are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this Reference Manual 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 M, ‘‘Glossary’’.

2. Lexical Elements

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. ♦ 1

2.1 Character Set

The only characters allowed outside of comments are the graphic_characters and format_effectors. ♦ 1

Syntax

character ::= graphic_character | format_effector | other_control_function 2

graphic_character ::= identifier_letter | digit | space_character | special_character 3

Static Semantics

The character repertoire for the text of an AVA program consists of the collection of characters specified in ISO 8859-1, ♦ plus a set of format_effectors and, in comments only, a set of other_control_functions; the coded representation for these characters is implementation defined (it need not be a representation defined within ISO-10646-1). 4

The description of the language definition in this Reference Manual uses the graphic symbols defined for Row 00: Basic Latin and Row 00: Latin-1 Supplement of the ISO 10646 BMP; these correspond to the graphic symbols of ISO 8859-1 (Latin-1); no graphic symbols are used in this Reference Manual for characters outside of Row 00 of the BMP. The actual set of graphic symbols used by an implementation for the visual representation of the text of an AVA program is not specified. 5

The categories of characters are defined as follows: 6

identifier_letter	upper_case_identifier_letter | lower_case_identifier_letter	7
upper_case_identifier_letter	Any character of Row 00 of ISO 10646 BMP whose name begins ‘‘Latin Capital Letter’’.	8
lower_case_identifier_letter	Any character of Row 00 of ISO 10646 BMP whose name begins ‘‘Latin Small Letter’’.	9
digit	One of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.	10
space_character	The character of ISO 10646 BMP named ‘‘Space’’.	11
special_character	Any character of the ISO 10646 BMP that is not reserved for a control function, and is not the space_character, an identifier_letter, or a digit.	12
format_effector	The control functions of ISO 6429 called character tabulation or tab (HT), ♦ carriage return (CR), line feed (LF), and form feed or page (FF).	13
other_control_function	Any control function, other than a format_effector, that is allowed in a comment; the set of other_control_functions allowed in comments is implementation defined.	14

The following names are used when referring to certain special_characters: 15

symbol	name	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	[	left square bracket
,	comma	]	right square bracket
–	hyphen-minus, minus	{	left curly bracket
.	full stop, dot, point	}	right curly bracket
/	solidus, divide		

♦

NOTES

17 1 ♦

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

2.2 Lexical Elements, Separators, and Delimiters

Static Semantics

1 The text of a program consists of the texts of one or more compilations. 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 The text of a compilation is divided into *lines*. In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than character tabulation (HT) signifies at least one end of line.

3 In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a space character, a format effector, or the end of a line, as follows:

4 • A space character is a separator except within a comment, a string_literal, or a character_literal.

5 • Character tabulation (HT) is a separator except within a comment.

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

7 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.

8 A *delimiter* is either one of the following special characters

9 & ’ ( ) * + , - . / : ; < = > |

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
♦	
<>	box

14

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. 15

2.3 Identifiers

Identifiers are used as names. 1

Syntax

```
identifier ::=
  identifier_letter {[underline] letter_or_digit}
```
2

letter_or_digit ::= identifier_letter | digit 3

An identifier shall not be a reserved word. 4

Abstract Syntax

Identifiers before overload resolution are just symbols. After overload resolution they have been uniquely identified by an integer index. **Uid**s are ‘‘unique identifiers’’. 5

sym ∈ Symbols		ACL2 predicate: SYMBOLP(*sym*)
id ∈ Id	== *sym* | sym_n	Identifier | Unique Identifier

Static Semantics

All characters of an identifier are significant, including any underline character. Identifiers differing only in the use of corresponding upper and lower case letters are considered the same. ♦ 6

♦

Examples

7 *Examples of identifiers:*

8 Count X Get_Symbol Ethelyn Marion

Snobol_4 X1 Page_Count Store_Next_Item

2.4 Numeric Literals

1 There ♦ is one kind of numeric_literal, ♦ *integer literals*. ♦ An integer literal is a numeric_literal ♦.

Syntax

2 numeric_literal ::= decimal_literal | based_literal

Abstract Syntax

3 All numeric and decimal literals are translated into simple integers, including based_numeric_literals.

$n \in$ N ACL2 predicate: INTEGERP(n)

NOTES

4 3 The type of an integer literal is *universal_integer*. ♦

2.4.1 Decimal Literals

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

Syntax

2 decimal_literal ::= numeral ♦ [exponent]

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

4 exponent ::= E + numeral | ♦

5 ♦

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.

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
♦
```

2.4.2 Based Literals

A based_literal is a numeric_literal expressed in a form that specifies the base explicitly. 1

Syntax

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

base ::= numeral 3

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

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

Legality Rules

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. 6

Static Semantics

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. 7

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

Examples

Examples of based literals: 9

10

```
2#1111_1111#   16#FF#     016#0ff#          -- integer literals of value 255
16#E#E1        2#1110_0000#                 -- integer literals of value 224
♦
```

2.5 Character Literals

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

Syntax

character_literal ::= ’graphic_character’ 2

Abstract Syntax

3

$c \in$ Character	== #\space ... #\~	CHARACTER-P(c) $\wedge$ STANDARD-CHAR-P(c)

NOTES

4 A character_literal is an enumeration literal of a character type. See 3.5.2. 4

Examples

5 *Examples of character literals:*

6 ’A’ ’*’ ’’’ ’ ’

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 ♦ values of a string type (see 4.2), and array subaggregates (see 4.3.3).

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.

Abstract Syntax

5

$s \in$ String	== "c^*"	STRINGP(s)

Static Semantics

6 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.

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

NOTES

8 5 An end of line cannot appear in a string_literal.

Examples

9 *Examples of string literals:*

10

```
"Message of the day:"

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

"Characters such as $, %, and } are allowed in string literals"
```

2.7 Comments

1 A comment starts with two adjacent hyphens and extends up to the end of the line unless the character immediately after the second hyphen is ‘|‘, in which case the text to the end of the line is part of an AVA annotation. See Section 3.12.

Syntax

comment ::= --{*non_end_of_line_*character} 2

If there are any *non_end_of_line_*characters the first one following the hyphens must not be a vertical bar. A comment may appear on any line of a program. 3

Static Semantics

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. 4

Examples

Examples of comments: 5

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

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 -- Removed

We might want to reintroduce pragmas to allow the restrictions of Annex H to be applied. But since Annex H doesn't provide enough leverage to get us to the AVA subset, pragmas remain moot.

2.9 Reserved Words

Syntax

1

2 The following are the *reserved words* (ignoring upper/lower case distinctions):

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

Syntax

3

4 The following are the AVA reserved words (ignoring upper/lower case distinctions): AVA reserved words only have meaning in the context of annotations.

assert	**fi**	**invariant**	**where**
axiom	**iff**	**isin**	
defun	**implies**	**theorem**	

NOTES

3 6 The reserved words appear in **lower case boldface** in this Reference Manual, except when used in the designator of an attribute (see 4.1.4). ♦ This is merely a convention — programs may be written in whatever typeface is desired and available.

2.10 Annotations -- New

1 Annotations allow the user to

2 • define functions, constants and theorems in the ACL2 logic [Kaufmann 94],

3 • assert logical specifications for AVA functions, procedures, types, and statements, and

4 • state axioms and purported theorems to be used in the analysis of programs.

See section 3.12 for further details on annotations.

5 Annotation lines start with two adjacent hyphens and extend up to the end of the line if the character immediately after the second hyphen is '|. Note that the first production for an annotation line violates the

syntax of 1.1.4(14) in that the ‘|‘ after the two hyphens is part of the production and not a production separator.

Syntax

annotation_line ::= --|{*non_end_of_line*_character} 6

Annotation lines are preprocessed by tools that recognize these special forms of Ada comments as AVA annotations.

3. Declarations and Types

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

3.1 Declarations

The language defines several kinds of named *entities* that are declared by declarations. The entity's *name* is defined by the declaration, usually by a defining_identifier, but sometimes by a defining_character_ literal ♦. 1

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

Syntax

```
basic_declaration ::=                                                   3
    type_declaration            | subtype_declaration
  | inner_declaration           | ♦
  | subprogram_declaration      | ♦
  | package_declaration         | renaming_declaration
  | axiom_decl
  | theorem_decl
  | defun_decl
  | ♦                           | ♦
  | ♦

defining_identifier ::= identifier                                      4

inner_declaration ::=                                                   5
    object_declaration
  | number_declaration
  | invariant_annotation
```

Abstract Syntax

6

$d_i \in$ InnerD	$== d_o \mid d_n \mid assert \mid invariant$
$d_L \in$ LogicDecl	$== defun \mid theorem \mid axiom$
$d \in$ Decl	$== d_i \mid subp \mid type \mid subtype \mid d_L$

Static Semantics

A *declaration* is a language construct that associates a name with (a view of) an entity. 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). AVA declarations also include *annotations* that further restrict the properties of declared objects and subprograms. 5

Each of the following is defined to be a declaration: any basic_declaration; an enumeration_literal_ specification; ♦ a component_declaration; a loop_parameter_specification; a parameter_specification; and a subprogram_body. ♦ 6

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 ♦ formal parameter names♦, or visibility to components of a type). In most cases, a 7 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)).

8 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). At such places the identifier is said to be a *name* of the entity (the direct_name or selector_name); the name is said to *denote* the declaration, the view, and the associated entity (see 8.6). The declaration is said to *declare* the name, the view, and in most cases, the entity itself.

9 As an alternative to an identifier, an enumeration literal can be declared with a character_literal as its name (see 3.5.1) ♦.

10 The syntax rules use the terms defining_identifier and defining_character_literal♦ for the defining occurrence of a name; these are collectively called *defining names*. The terms direct_name and selector_name are used for usage occurrences of identifiers and character_literals♦. These are collectively called *usage names*.

Dynamic Semantics

11 The process by which a construct achieves its run-time effect is called *execution*. This process is also called *elaboration* for declarations and annotations and *evaluation* for expressions. One of the terms execution, elaboration, or evaluation is defined by this Reference Manual for each construct that has a run-time effect.

NOTES

12 1 At compile time, the declaration of an entity *declares* the entity. At run time, the elaboration of the declaration *creates* the entity.

3.2 Types and Subtypes

Static Semantics

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

2 Types are grouped into *classes* of types, reflecting the similarity of their values and primitive operations. There exist several *language-defined classes* of types (see NOTES below). *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

3 The elementary types are the *discrete scalar* types♦. Discrete types are either *integer* types or are defined by enumeration of their values (*enumeration* types). ♦

4 The composite types are the *record* types ♦ and *array* types♦ A *private* type ♦ represents a partial view (see 7.3) of a type, providing support for data abstraction. A partial view is a composite type. ♦

5 ♦

The term *subcomponent* is used in this Reference Manual 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. Similarly, a *part* of an object or value is used to mean the whole object or value, or any set of its subcomponents. 6

The set of possible values for an object of a given type can be subjected to a condition that is called a *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, and 3.6.1 for index_constraints♦. 7

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. Similarly, the associated constraint is called the constraint *of* the subtype. The set of values of a subtype consists of the values of its type that satisfy its constraint. Such values *belong* to the subtype. 8

A subtype is called an *unconstrained* subtype if ♦ its type allows range or index♦ 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

NOTES

2 ♦ Only certain 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)♦. The following are examples of ‘‘interesting’’ *language-defined classes*: elementary, scalar, discrete, enumeration, character, boolean, integer, ♦ composite, array, string, ♦ record♦. Special syntax is provided to define types in each of these classes. 10

These language-defined classes are organized like this: 11

12

```
all types
    elementary
        scalar
            discrete
                enumeration
                    character
                    boolean
                    other enumeration
                integer
                    signed integer
♦
    composite
        array
            string
            other array
        ♦ record
        ♦
```

♦ 13

3.2.1 Type Declarations

A type_declaration declares a type and its first subtype. 1

Syntax

```
type_declaration ::=  full_type_declaration
   | ♦
   | private_type_declaration
   | ♦
```
2

3
```
full_type_declaration ::=
    type defining_identifier ♦ is type_definition;
  | ♦
```

4
```
type_definition ::=
    enumeration_type_definition    | ♦
  | ♦                              | array_type_definition
  | record_type_definition         | ♦
  | ♦
```

Legality Rules

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

Abstract Syntax

6

type ∈ Type	== $type_r$ | $type_a$ | $type_e$ | *id* | *range*
d_t ∈ TypeDecl	== **type** *id* [*type*]

Static Semantics

7 The defining_identifier of a type_declaration denotes the *first subtype* of the type. ♦ The remainder of the type_declaration defines the remaining characteristics of (the view of) the type.

8 A type defined by a type_declaration is a *named* type; such a type has one or more nameable subtypes. For a named type whose first subtype is T, this Reference Manual sometimes refers to the type of T as simply ‘‘the type T.’’

9 A named type that is declared by a full_type_declaration, ♦ is called a *full type*. The type_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.

10 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’’.

11 The *predefined types* (for example the types Boolean, ♦ 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.

Dynamic Semantics

12 The elaboration of a full_type_declaration consists of the elaboration of the full type definition. Each elaboration of a full type definition creates a distinct type and its first subtype.

Examples

13 *Examples of type definitions:*

14
```
(White, Red, Yellow, Green, Blue, Brown, Black)
range 1 .. 72
array(index) of Integer
```

15 *Examples of type declarations:*

16
```
type Color  is (White, Red, Yellow, Green, Blue, Brown, Black);
♦
type Table  is array(index) 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 Table is sometimes used in this Reference Manual to refer to the type of Table, where Table denotes the first subtype of the type. ♦ 17

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 subtype_indication ::= subtype_mark [constraint]

4 subtype_mark ::= *subtype*_name

5 constraint ::= scalar_constraint | composite_constraint

6 scalar_constraint ::=
  range_constraint | ♦

7 composite_constraint ::=
  index_constraint | ♦

Abstract Syntax

8

con ∈ Constraints	== *subtype* | **unconstrained** | *range* | **attr** *id* **range**	
tm ∈ TM	== **type-mark** *id con*	
subtype ∈ Subtype	== *id* | *tm*	A subtype_indication
d_s ∈ SubtypeDecl	== **subtype** *id subtype*	

Name Resolution Rules

9 A subtype_mark shall resolve to denote a subtype. The type *determined by* a subtype_mark is the type of the subtype denoted by the subtype_mark.

Dynamic Semantics

10 The elaboration of a subtype_declaration consists of the elaboration of the subtype_indication. 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:

- 11 The constraint is first elaborated.
- 12 A check is then made that the constraint is *compatible* with the subtype denoted by the subtype_mark.

13 The condition imposed by a constraint is the condition obtained after elaboration of the constraint. 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. The exception Constraint_Error is raised if any check of compatibility fails.

NOTES

14 4 A scalar_constraint may be applied to a subtype of an appropriate scalar type (see 3.5), even if the subtype is already constrained. On the other hand, a composite_constraint may be applied to a composite subtype ♦ only if the composite subtype is unconstrained (see 3.6.1).

Examples

15 *Examples of subtype declarations:*

16
```
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 Square    is Matrix(1 .. 10, 1 .. 10);       -- see 3.6
♦
```

3.2.3 Classification of Operations

Static Semantics

1 An operation *operates on a type T* if it yields a value of type *T*, or if it has an operand whose expected type (see 8.6) is *T*♦. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined operation* of the type. The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms. ♦

2 The *primitive subprograms* of a specific type are defined as follows:

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

4 • ♦

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 • ♦

♦

8 A primitive subprogram whose designator is an operator_symbol is called a *primitive operator*.

3.3 Objects and Named Numbers

1 Objects are created at run time and contain a value of a given type. An object can be created and initialized as part of elaborating a declaration, evaluating an ♦ aggregate or 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 All of the following are objects:

3 • the entity declared by an object_declaration;

4 • a formal parameter of a subprogram♦;

5 • ♦

- a loop parameter; 6
- ♦ 7
- ♦ 8
- ♦ 9
- the result of evaluating a function_call (or the equivalent operator invocation); 10
- the result of evaluating an aggregate; 11
- a component♦ of another object. 12

13 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 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. 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.

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:

- an object declared by an object_declaration with the reserved word **constant**; 16
- a formal parameter ♦ of mode **in**; 17
- ♦ 18
- a loop parameter♦; 19
- ♦ 20
- the result of evaluating a function_call or an aggregate; 21
- a selected_component or indexed_component ♦ of a constant. ♦ 22

23 At the place where a view of an object is defined, a *nominal subtype* is associated with the view. 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*. A subtype is an indefinite subtype if it is an unconstrained array subtype♦; 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 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 ♦ an array object, if the nominal subtype is constrained then so is the actual subtype.

3.3.1 Object Declarations

1 An object_declaration declares a *stand-alone* object with a given nominal subtype and♦ an explicit initial value given by an initialization expression. ♦

Syntax

2
```
object_declaration ::=
   defining_identifier_list : [constant] subtype_indication [:= expression];
 | ♦
 | ♦
 | ♦
```

3
```
defining_identifier_list ::= defining_identifier {, defining_identifier}
```

Abstract Syntax

4

mode ∈ Mode	== **constant** | **variable**
d_o ∈ ObjectDecls	== **object** *id mode subtype* [*expr*]

Name Resolution Rules

5 ♦ The type expected for the expression following the compound delimiter := is that of the object. This expression is called the *initialization expression*.

Legality Rules

6 An object_declaration without the reserved word **constant** declares a variable object. ♦

Static Semantics

7 An object_declaration with the reserved word **constant** declares a constant object. If it has an initialization expression, then it is called a *full 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.

8 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 Reference Manual relies on this equivalence; explanations are given for declarations with a single defining_identifier.

9 The subtype_indication ♦ of an object_declaration defines the nominal subtype of the object. The object_declaration declares an object of the type of the nominal subtype. ♦

Dynamic Semantics

10 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 ♦ of its initial value; the object is said to be *constrained by its initial value*. ♦ An explicit initial value is required. When not constrained by its initial value, the actual and nominal subtypes of the object are the same. If its actual subtype is constrained, the object is called a *constrained object*.

♦ 11

- ♦ 12
- ♦ 13
- ♦ 14
- ♦ 15

The elaboration of an object_declaration proceeds in the following sequence of steps: 16

1. The subtype_indication♦ is first elaborated. This creates the nominal subtype♦. 17
2. ♦ 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). 18
3. The object is created♦. ♦ 19
4. ♦ Initial values ♦ are assigned to the object ♦. ♦ 20

♦ ♦ 21

♦ 22

NOTES

7 ♦ 23

8 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♦. An object declared by a loop_parameter_specification♦ or parameter_specification ♦ is not called a stand-alone object. 24

9 ♦ 25

Examples

Example of a multiple object declaration: 26

-- *the multiple object declaration* 27

```
♦                                                            28
Buick, Ford: CAR := (Number => 30300, Owner => "Smith, Michael K. ");
```

-- *is equivalent to the two single object declarations in the order given* 29

```
♦                                                            30
Buick: CAR := (Number => 30300, Owner => "Smith, Michael K. ");
Ford: CAR := (Number => 30300, Owner => "Smith, Michael K. ");
```

Examples of variable declarations: 31

```
♦                                                            32
Size        : Integer range 0 .. 10_000 := 0;
Sorted      : Boolean := False;
♦
Hello       : constant String := "Hi, world.";
```

Examples of constant declarations: 33

```
Limit     : constant Integer := 10_000;                      34
Low_Limit : constant Integer := Limit/10;
♦
```

3.3.2 Number Declarations

1 A number_declaration declares a named number.

Syntax

2
```
number_declaration ::=
   defining_identifier_list : constant := static_expression;
```

Name Resolution Rules

3 The *static*_expression given for a number_declaration is expected to be of any numeric type.

Legality Rules

4 The *static*_expression given for a number declaration shall be a static expression, as defined by clause 4.9.

Abstract Syntax

5 $d_n \in$ NumberDecl == **number** *id mode expr*

Static Semantics

6 The named number denotes a value of type *universal_integer*♦.

7 The value denoted by the named number is the value of the *static*_expression, converted to the ♦ type *universal_integer*.

Dynamic Semantics

8 The elaboration of a number_declaration has no effect.

Examples

9 *Examples of number declarations:*

10 ♦

11
```
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
```

♦

3.4 Derived Types and Classes -- Largely Removed

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 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).

3 Every type is either a *specific* type ♦ or a *universal* type. A specific type is one defined by a type_ declaration♦. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

Universal types — Universal types are defined for (and belong to) the integer ♦ class, and are referred to in this standard as ♦ *universal_integer* ♦. ♦ A value of a universal type (including an integer ♦) is ‘‘universal’’ in that it is acceptable where some particular type in the class is expected (see 8.6). 6

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. ♦ 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

The integer ♦ class ♦ has a specific root type in addition to its universal type, named ♦ *root_integer* ♦. 8

A ♦ universal type is said to *cover* all of the types in its class. A specific type covers only itself. 9

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*. ♦ The universal types are defined to be descendants of the root types of their classes. If a type *T2* is a descendant of a type *T1*, then *T1* is called an *ancestor* of *T2*. ♦ 10

♦ 11

NOTES

10 Because operands of a universal type are acceptable to the predefined operators of any type in their class, ambiguity can result. For *universal_integer* ♦, this potential ambiguity is resolved by giving a preference (see 8.6) to the predefined operators of the corresponding root type (*root_integer* ♦). Hence, in an apparently ambiguous expression like 12

```
1 + 4 < 7
```
13

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.

3.5 Scalar Types

Scalar types comprise enumeration types and integer types♦. Enumeration types and integer types are called *discrete* types; each value of a discrete type has a *position number* which is an integer value. Integer types ♦ are called *numeric* types. All scalar types are ordered, that is, all relational operators are predefined for their values. 1

Syntax

range_constraint ::= **range** range 2

range ::= range_attribute_reference 3
  | simple_expression .. simple_expression

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’’. 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. 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. A value *satisfies* a range constraint if it belongs to the associated range. One range is *included* in another if all values that belong to the first range also belong to the second. ♦ 4

Abstract Syntax

5

from, to == *n* | *c* | *id*

range ∈ Range == **range** *from to*

Name Resolution Rules

6 For a subtype_indication containing a range_constraint, ♦ the type of the range shall resolve to that of the type determined by the subtype_mark of the subtype_indication. 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.

Static Semantics

7 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. ♦

8 A constrained scalar subtype is one to which a range constraint applies. 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

9 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. A range_constraint is *compatible* with a scalar subtype if and only if its range is compatible with the subtype.

10 The elaboration of a range_constraint consists of the evaluation of the range. 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. If a range_attribute_reference is given, the evaluation of the range consists of the evaluation of the range_attribute_reference.

11 *Attributes*

12 For every scalar subtype S, the following attributes are defined:

13 S'First — S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S.

14 S'Last — S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S.

15 ♦

16 S'Base — S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the *base subtype* of the type.

17 ♦

18 ♦

22 S'Succ — S'Succ denotes a function with the following specification:

23
```
function S'Succ(Arg : S'Base)
  return S'Base
```

24 For an enumeration type, the function returns the value whose position number is one more than that of the value of *Arg*; 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*. ♦ Constraint_Error is raised if there is no such machine number. ♦

S'Pred S'Pred denotes a function with the following specification: 25

```
function S'Pred(Arg : S'Base)                                        26
  return S'Base
```

For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; 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*. ♦ Constraint_Error is raised if there is no such machine number. ♦ 27

♦ 28

S'Image S'Image denotes a function with the following specification: 35

```
function S'Image(Arg : S'Base)                                       36
  return String
```

The function returns an image of the value of *Arg* as a String. 37

The lower bound of the result is one. 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. 38

The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); 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 or implementation-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). ♦ 39

♦ 40

♦ 41

S'Value S'Value denotes a function with the following specification: 52

```
function S'Value(Arg : String)                                       53
  return S'Base
```

This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces. 54

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; otherwise Constraint_Error is raised. For the evaluation of a call on S'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), and the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint_Error is raised. ♦ 55

56

NOTES

19 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

20 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

59 21 For any value V (including any nongraphic character) of an enumeration subtype S, S’Value(S’Image(V)) equals V, ♦. Neither expression ever raises Constraint_Error.

Examples

60 *Examples of ranges:*

61
```
-10 .. 10
X .. X + 1
♦
Red .. Green      -- see 3.5.1
1 .. 0            -- a null range
♦
```

62 *Example of range constraints:*

63
```
♦
range S’First+1 .. S’Last-1
```

3.5.1 Enumeration Types

1 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.

Abstract Syntax

6

$e \in$ EnumLiteral	== *id* | *c*
$type_e \in$ EnumerationType	== **enum** e^*

Static Semantics

7 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.

8 Each enumeration literal corresponds to a distinct value of the enumeration type, and to a distinct position number. 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.

9 The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.

10 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

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. ♦ 11

When called, the parameterless function associated with an enumeration literal returns the corresponding value of the enumeration type. 12

NOTES

22 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). 13

Examples

Examples of enumeration types and subtypes: 14

```
type Day    is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);                    15
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

type Hexa   is ('A', 'B', 'C', 'D', 'E', 'F');                         16
type Mixed  is ('A', 'B', '*', B, None, '?', '%');

subtype Weekday is Day   range Mon .. Fri;                             17
subtype Major   is Suit  range Hearts .. Spades;
subtype Rainbow is Color range Red .. Blue;  -- the Color Red, not the Light
```

3.5.2 Character Types

Static Semantics

An enumeration type is said to be a *character type* if at least one of its enumeration literals is a character_literal. 1

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 10646 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 non-graphic 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 and ♦Value; these names are given in the definition of type Character in A.1, ‘‘The Package Standard’’, but are set in *italics*. 2

♦ 3

Implementation Permissions

In a nonstandard mode, an implementation may provide other interpretations for the predefined type Character ♦, to conform to local conventions. 4

♦

Examples

Example of a character type: 8

```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');               9
```

3.5.3 Boolean Types

Static Semantics

1 There is a predefined enumeration type named Boolean, declared in the visible part of package Standard. It has the two enumeration literals False and True ordered with the relation False < True. ♦ The predefined type Boolean is called a *boolean* type. ♦

Abstract Syntax

2 $b \in$ BooleanLiteral == **true** | **false**

3.5.4 Integer Types

1 ♦

Abstract Syntax

2 The predefined integer type in standard is $integer_0$.

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. ♦

9 ♦

11 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.

12 Integer has two predefined subtypes, declared in the visible part of package Standard:

13
```
    subtype Natural  is Integer range 0 .. Integer’Last;
    subtype Positive is Integer range 1 .. Integer’Last;
```

14 ♦ *Root_integer* is an anonymous predefined (specific) integer type, whose base range is System.Min_Int .. System.Max_Int. ♦ Integer literals are all of the type *universal_integer*, the universal type for the class rooted at *root_integer*, allowing their use with the operations of any integer type. ♦

15 The *position number* of an integer value is equal to the value.

16 ♦

Dynamic Semantics

17 ♦

19 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.

20 For any integer type, Constraint_Error is raised by the operators "/", "**rem**", and "**mod**" if the right operand is zero.

Implementation Requirements

In an implementation, the range of Integer shall include the range –2**15+1 .. +2**15–1. The smallest (most negative) value supported by the predefined integer types of an implementation (excluding *universal_integer*) is the named number AVA.Min_Int and the largest (most positive) value is AVA.Max_Int. An implementation must **not** accept a compilation unit containing a static *univeral_integer* expression whose value lies outside of the range AVA.Min_Int .. AVA.Max_Int.[2] 21

♦ 22

System.Max_Binary_Modulus shall be at least 2**16. 23

♦

NOTES
24 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

25 ♦ 31

Examples

Examples of integer ♦ subtypes: 32

♦ 33

34
```
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;
```

♦ 34

♦

3.5.5 Operations of Discrete Types

Some of the operations of a discrete type require or return information about the constraints of the subtype or have names dependent on the subtype name. In this case we talk about operations or attributes *of the subtype*. Formally, these are operations of the *base* type that may take additional, subtype dependent arguments to express constraint information.

Static Semantics

For every discrete subtype S, the following attributes are defined: 1

S’Pos S’Pos denotes a function with the following specification: 2

```
function S'Pos(Arg : S'Base)
  return universal_integer
```
3

[2]**IMPLEMENTATION REQUIREMENT**. Ada requires that such expressions *be accepted*, unless insufficient resources (memory) are available. We require otherwise in order that:

1. we have a single, predictable model of arithmentic operations and
2. we can write down a requirement that will allow us to prove whether or not an expression is static. □

Note that we have deleted the permission to return a value outside of the base range. This means that the optimization of ((Ada.Max_Int+1)-1) to Ada.Max_Int is not permissible.

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 :   integer)
  return S'Base
```

7 This function returns a value of the type of S whose position number equals the value of *Arg*. 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. ♦

♦

NOTES

9 28 Indexing and loop iteration use values of discrete types.

10 29 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♦ 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.

11 30 ♦

12 31 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:

13
```
S'Val(S'Pos(X)) = X
S'Pos(S'Val(N)) = N
```

Examples

14 *Examples of attributes of discrete subtypes:*

15 -- *For the types and subtypes declared in subclause 3.5.1 the following hold:*

16
```
--  Color'First   = White,   Color'Last   = Black
--  Rainbow'First = Red,     Rainbow'Last = Blue
```

16
```
--  Color'Succ(Blue) = Rainbow'Succ(Blue) = Brown
--  Color'Pos(Blue)  = Rainbow'Pos(Blue)  = 4
--  Color'Val(0)     = Rainbow'Val(0)     = White
```

♦

3.5.6 Real Types -- Removed

3.5.7 Floating Point Types -- Removed

3.5.8 Operations of Floating Point Types -- Removed

3.5.9 Fixed Point Types -- Removed

3.5.10 Operations of Fixed Point Types -- Removed

3.6 Array Types

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 integer types. The value of an array object is a composite value consisting of the values of the components. 1

Syntax

array_type_definition ::=
  unconstrained_array_definition | constrained_array_definition 2

unconstrained_array_definition ::=
  array(index_subtype_definition {, index_subtype_definition}) **of** component_definition 3

index_subtype_definition ::= subtype_mark **range** <> 4

constrained_array_definition ::=
  array (*integer*_subtype_definition {, *integer*_subtype_definition}) **of** component_definition 5

discrete_subtype_definition ::= *discrete*_subtype_*mark* | range 6

integer_subtype_definition ::= ***integer*_**subtype_*mark* | range 7

component_definition ::= ♦ subtype_indication 8

Abstract Syntax

9

$type_a \in$ ArrayType == **array** *tm type*

Name Resolution Rules

For an integer_subtype_definition that is a range, the range shall resolve to be of some specific integer 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). 8

Legality Rules

Each index_subtype_definition or integer_subtype_definition in an array_type_definition defines an *index subtype*; its type (the *index type*) shall be of an integer type. 9

The subtype defined by the subtype_indication of a component_definition (the *component subtype*) shall be a definite subtype. } 10

♦ 11

Static Semantics

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 13 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; this range of values is called the *index range*. The *bounds* of an array are the bounds of its index ranges. 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). The *length* of a one-dimensional array is the length of its only dimension.

14 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).

15 An unconstrained_array_definition defines an array type with an unconstrained first subtype. Each integer_subtype_definition defines the corresponding index subtype to be the subtype denoted by the subtype_mark. 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).

16 A constrained_array_definition defines an array type with a constrained first subtype. Each integer_subtype_definition defines the corresponding index subtype, as well as the corresponding index range for the constrained first subtype. The *constraint* of the first subtype consists of the bounds of the index ranges. ♦

17 The discrete subtype defined by a discrete_subtype_definition or an integer_subtype_definition is either that defined by the subtype_*mark*, or a subtype determined by the range as follows:

18 - If the type of the range resolves to *root_integer*, then the subtype definition defines a subtype of the predefined type Integer with bounds given by a conversion to Integer of the bounds of the range; ♦

19 - Otherwise, the index_range defines a subtype of the type of the range, with the bounds given by the range.

20 The component_definition of an array_type_definition defines the nominal subtype of the components. ♦

Dynamic Semantics

21 The elaboration of an array_type_definition creates the array type and its first subtype, and consists of the elaboration of any index_ranges and the component_definition.

22 The elaboration of an *integer*_subtype_definition creates the integer subtype, and consists of the elaboration of the subtype_indication or the evaluation of the range. The elaboration of a component_definition in an array_type_definition consists of the elaboration of the subtype_indication. The elaboration of any *integer*_subtype_definitions and the elaboration of the component_definition are performed in an arbitrary order.

NOTES

23 41 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 42 Each elaboration of an array_type_definition creates a distinct array type. ♦

Examples

25 *Examples of type declarations with unconstrained array definitions:*

26
```
♦
type Matrix     is array(Integer  range <>, Integer range <>) of Integer;
type Bit_Vector is array(Integer  range <>) of Boolean;
type Roman      is array(Positive range <>) of Roman_Digit; -- see 3.5.2
```

Examples of type declarations with constrained array definitions: 27

28
```
type Table    is array(1 .. 10) of Integer;
type Schedule is array(1 .. 7) of Boolean;
type Line     is array(1 .. Max_Line_Size) of Character;
```

Examples of object declarations with array type definitions: 29

30
```
Bv : Bit_Vector(0..7) := (others => false);
♦
Tuple : array(Positive range <>) of Integer := (1, 2, 3); ♦
```

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

The type of an discrete_range is the type of the subtype defined by the subtype_indication, or the type of the range. For an index_constraint, each discrete_range shall resolve to be of the type of the corresponding index and thus must be an integer range. 4

Legality Rules

An index_constraint shall appear only in a subtype_indication whose subtype_mark denotes an unconstrained array subtype♦; the index_constraint shall provide a discrete_range for each index of the array type. 5

Static Semantics

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

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. If any of the discrete_ranges defines a null range, any array thus constrained is a *null array*, having no components. 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

The elaboration of an index_constraint consists of the evaluation of the discrete_range(s), in an arbitrary order. The evaluation of a discrete_range consists of the elaboration of the subtype_indication or the evaluation of the range. 8

NOTES
43 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). 9

10 44 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)  := (others => 0);  -- see 3.6
Rectangle : Matrix(1 .. 20, 1 .. 30) := (others => 0);
Inverse   : Matrix(1 .. N,  1 .. N)  := (others => 0);  -- N need not be static
```

13
```
Filter    : Bit_Vector(0 .. 31) := (others => true);
```

14 *Example of array declaration with a constrained array subtype:*

15
```
My_Schedule : Schedule := (others => false);  -- all arrays of type Schedule have the same bounds
```

16 ♦

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 type universal_integer. The value of N shall be positive (nonzero) and no greater than the dimensionality of the array.

Static Semantics

2 The following attributes are defined for a prefix A that is of an array type ♦, or denotes a constrained array subtype: ♦

3	A'First	A'First denotes the lower bound of the first index range; its type is the corresponding index type.
4	A'First(N)	A'First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type.
5	A'Last	A'Last denotes the upper bound of the first index range; its type is the corresponding index type.
6	A'Last(N)	A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type.
7	A'Range	A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once.
8	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.
9	A'Length	A'Length denotes the number of values of the first index range (zero for a null range); its type is *universal_integer*.
10	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*.

Implementation Advice

11 An implementation should normally represent multidimensional arrays in row-major order, consistent with the notation used for multidimensional array aggregates (see 4.3.3). ♦

NOTES

12 45 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:

```
    A'Length(N) = A'Last(N) - A'First(N) + 1
```
13

46 ♦ 14

47 The predefined operations of an array type include ♦ the predefined equality operators. For a one-dimensional array type, they include the predefined concatenation operators♦ 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

48 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♦. 16

Examples

Examples (using arrays declared in the examples of subclause 3.6.1): 17

```
--  Filter'First       =    0   Filter'Last        =  31   Filter'Length  =  32
--  Rectangle'Last(1)  =   20   Rectangle'Last(2)  =  30
```
18

3.6.3 String Types

Static Semantics

A one-dimensional array type whose component type is a character type is called a *string* type. 1

There is one predefined string type, String ♦, indexed by values of the predefined subtype Positive; these are declared in the visible part of package Standard: 2

```
subtype Positive is Integer range 1 .. Integer'Last;
```
3

4
```
type String is array(Positive range <>) of Character;
♦
```

NOTES
49 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♦ one-dimensional array types. The ordering operators <, <=, >, and >= are predefined for string types♦; these ordering operators correspond to lexicographic order (see 4.5.2). 4

Examples

Examples of string objects: 5

```
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)
```
6

7
```
Ask_Twice  : String  := Question & Question; -- constrained to (1..40)
Ninety_Six : constant Roman   := "XCVI";       -- see 3.5.2 and 3.6
```

♦

3.7 Discriminants -- Removed

3.8 Record Types

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. 1

Syntax

```
2        record_type_definition ::= ♦ record_definition

3        record_definition ::=
           record
             component_list
           end record
          | ♦

4        component_list ::=
             component_item {component_item}
          | ♦
          | null;

5        component_item ::= component_declaration♦

6        component_declaration ::=
           defining_identifier_list : component_definition ♦;
```

Abstract Syntax

7

fs ∈ FieldSpec	== **fs** *id type*
$type_r$ ∈ RecordType	== **record** fs^*

♦

Legality Rules

8 ♦

9 Each component_declaration declares a *component* of the record type. ♦ The identifiers of all components of a record type shall be distinct.

10 ♦

Static Semantics

14 The component_definition of a component_declaration defines the (nominal) subtype of the component. ♦

15 ♦

Dynamic Semantics

16 The elaboration of a record_type_definition creates the record type and its first subtype, and consists of the elaboration of the record_definition. The elaboration of a record_definition consists of the elaboration of its component_list, if any.

17 The elaboration of a component_list consists of the elaboration of the component_items ♦ in the order in which they appear. The elaboration of a component_declaration consists of the elaboration of the component_definition.

18 ♦ For the elaboration of a component_definition of a component_declaration, ♦ the subtype_indication is elaborated. ♦

NOTES

55 A component_declaration with several identifiers is equivalent to a sequence of single component_declarations, as explained in 3.3.1. 19

56 ♦ 20

57 The subtype defined by a component_definition (see 3.6) has to be a definite subtype. 21

58 ♦ The same components of a record type are present in all values of the type. 22

59 ♦ 23

60 The predefined operations of a record type include membership tests, qualification, ♦ and the predefined equality operators. 24

61 A component of a record can be named with a selected_component. A value of a record can be specified with a record_aggregate♦. 25

Examples

Examples of record type declarations: 26

27
```
type Date is
   record
      Day   : Integer range 1 .. 31;
      Month : Month_Name;
      Year  : Integer range 0 .. 4000;
   end record;
```

28
```
type Rational is
  record
    num : Integer;
    den : Integer;
  end record;
```

29
```
type Car is
  record
    Number : Integer;
    Owner  : String(1 .. 20);
  end record;
type Person is
  record
    Name    : String(1 .. 20);
    Birth   : Date;
    Age     : Integer range 0 .. 130;
    Vehicle : Car;
    Spouse  : String(1 .. 20);
  end record;
```

Examples of record variables: 29

30
```
Tomorrow, Yesterday : Date (20, 5, 49);
A, B, C : Rational := (1,1);
♦

Next_Car : Car := (34549821, "Smith, Michael K.   ");
```

31
```
Next_Person : Person := ("Smith, Michael K.   ", Yesterday, 40, Next_Car, "Smith, Elizabeth B. ")
```

3.8.1 Variant Parts and Discrete Choices -- Removed

3.9 Tagged Types and Type Extensions -- Removed

3.10 Access Types -- Removed

3.11 Declarative Parts

1 A declarative_part contains declarative_items (possibly none).

Syntax

2 declarative_part ::= {declarative_item}

3 declarative_item ::=
basic_declarative_item | body

4 basic_declarative_item ::=
basic_declaration | ♦

5 body ::= proper_body | ♦

6 proper_body ::=
subprogram_body | package_body | ♦

7 **inner_declarative_part**::= {**inner_declaration**}

Dynamic Semantics

8 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. The elaboration of an inner_declarative_part consists of the elaboration of the inner_declarative_items, if any, in the order in which they are given in the inner_declarative_part.

9 An elaborable construct is in the *elaborated* state after the normal completion of its elaboration. Prior to that, it is *not yet elaborated*.

10 For a construct that attempts to use a body, a check (Elaboration_Check) is performed, as follows:

11 • For a call to a ♦ subprogram ♦, a check is made that the subprogram_body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

12 • ♦

13 • ♦

14 • ♦

15 The exception Program_Error is raised if any of these checks fails.

3.11.1 Completions of Declarations

1 Declarations sometimes come in two parts. A declaration that requires a second part is said to *require completion*. The second part is called the *completion* of the declaration (and of the entity declared), and is either another declaration, or a body♦.

Name Resolution Rules

A construct that can be a completion is interpreted as the completion of a prior declaration only if: 2

- The declaration and the completion occur immediately within the same declarative region; 3
- The defining name or defining_program_unit_name in the completion is the same as in the declaration♦; 4
- If the declaration is overloadable, then the completion has a type-conformant profile♦. 5

Legality Rules

An implicit declaration shall not have a completion. For any explicit declaration that is specified to *require completion*, there shall be a corresponding explicit completion. 6

At most one completion is allowed for a given declaration. Additional requirements on completions appear where each kind of completion is defined. 7

Static Semantics

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

NOTES
62 ♦ 9

63 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♦. The rules of 13.14, ‘‘Freezing Rules’’, prevent , at compile time, premature uses of other entities such as private types and deferred constants. 10

3.12 Annotation Declarations -- New

Annotations allow the user to 1

- define functions, constants and theorems in the ACL2 logic [Kaufmann 94], 2
- assert logical specifications for AVA functions, procedures, objects, and statements, 3
- and state axioms and purported theorems to be used in the analysis of programs. 4

One difficulty that the user may encounter is that expressions in ACL2 and AVA have different semantics. For example, ‘‘+’’ in AVA text is an operation subject to the normal rules of Ada, e.g. the exception CONSTRAINT_ERROR is raised if the result is too large. The ‘‘+’’ in ACL2 is a total function, identical to mathematical plus in the case that its arguments are integers. Another difference is that variables in ACL2 are untyped. We may assert restrictions on their values in various contexts by virtue of predicates, e.g. ‘‘integerp(x)’’. 5

Communication between the AVA world and the logic is by virtue of assertions on the value of AVA variables in the current environment. 6

Syntax

assert_annotation ::= **assert** logical_expression ; 7

invariant_annotation ::= **invariant** logical_expression ; 8

transition_annotation ::= **where** logical_expression ; 9

10 subprogram_annotation ::=
  where logical_expression
  | **where return** [identifier ,] logical_expression

11 axiom_decl ::= **axiom** identifier logical_expression ;

12 theorem_decl ::= **theorem** identifier logical_expression ;

13 defun_decl ::= **defun** identifier arglist logical_expression ;

14 arglist ::= ({identifier})

Annotation lines are preprocessed by tools that recognize these special forms of Ada comments as AVA annotations. The syntax was originally more similar to that of ANNA [Luckham 90]. As we faced the issue of providing a precise semantics for annotations in ACL2 they have evolved away from ANNA.

Abstract Syntax

15

assert ∈ Assert	== **assert** *lexpr*
invariant ∈ Invariant	== **invariant** *lexpr*
transition ∈ Transition	== **transtion** *lexpr*
$spec_{ret}$ ∈ ReturnRelation	== **spec-return** *sym lexpr*
$spec_{val}$ ∈ ReturnValue	== **spec-value** *lexpr*
$spec_{p}$ ∈ SubprogramAnnotation	== $spec_{ret}$ | $spec_{val}$ | *transition*
axiom ∈ Axiom	== **axiom** *sym lexpr*
theorem ∈ Theorem	== **theorem** *sym lexpr*
defun ∈ Defun	== **defun** *sym* sym^* *lexpr*

Formal Static Semantics

16 Annotations have *scope* and *points of application*. Within the scope of the annotation it is evaluated with respect to an output state (typically the current state) and possibly an input state whenever the computation reaches a point of application.

17 The scope of an assert_annotation appearing in a statement list is the annotation itself and its point of application is that of the assert statement in the statement list.

18 The scope of a transition_annotation is the immediately preceding statement. The input state is the state before execution of the statement. The output state is the current state. The point of application is immediately after the statement.

19 The scope of an invariant_annotation appearing in a declaration list is the scope of the enclosing declarative region and its points of application follow every statement in the declarative region.

20 The scope of a subprogram_annotation is identical to the scope of its associated subprogram_declaration. The points of application are the returns from every call on the designated subprogram. The input state is the state immediately preceding the evaluation of the body of the subprogram. The output state is the state at the point of return from the body of the subprogram. Thus, the annotation will normally be stated in terms of the formal parameters of the subprogram.

21 A Defun_decl defines a *specification* function in the logic of ACL2 to be used within annotations. We expect most annotations to be stated in terms of these functions, as opposed to the approach in ANNA, where virtual function definitions are used. The advantage is that we can provide a meaning in the logic for such functions. An attempt to state properties in terms of the AVA executable functions is possible,

but is indirect and would take the form of a statement about the application of the operational semantics to such a function in some specific environment.

Formal Dynamic Semantics

Properties of *specification* functions can be described by theorem_decls and axiom_decls. Axioms are *assumptions*. Theorems are conjectures to be *proven*. These have no effect on the evaluation of an AVA program. 22

The meaning given to annotations depends on their scope and points of application. 23

Annotation evaluation proceeds as follows. The logical variable ‘‘env’’ contains a mapping from program variable names to values. The annotation is evaluated according to the rules of the the ACL2 logic and the interpretation provided for logical_expressions (see section 4.10. If the result is non-false (in the ACL2 sense), computation proceeds. If the result is false (NIL), the exception logical_error is raised. Logical_error cannot be handled. Proving the correctness of a program or subprogram requires proving the absence of such exceptions. 24

Examples

Example of a compound statement invariant. 25

If *x* is less than or equal to *y* when we enter the loop, then *x* = *y* when we exit. 26

```
while x < y loop                                   27
   x := x+1;
  end loop;
--| where if in(@x le @y) then @x = @y fi ;
```

Example of an assert_annotation.

```
--| assert @x < 5 and @y < 10;                     28
  x := x * y;
--| assert @x < 50;
```

Examples of function annotation.

‘‘Get’’ is predefined to select an array or record element from a literal. ‘‘pattern_ok’’ would be defined in ACL2 via a defun_decl. 29

```
function filter_table_ok (table : a_filter_table) return boolean;      30
--| where return
--|   (all i in (1 .. filter_max) , pattern_ok(get(@table,i)));
```

In the following example we are asserting that the value that f returns, denoted by *z*, when f is evaluated in environment *env* is equal to the result of evaluating ‘‘g(value(’x, env),value(’y, env))’’ in the logic. 31

```
function f(x,y:T1) return T2;                      32
--| where return z , g(@x, @y) = z;
```

We could state this more simply by leaving out the variable, *z*. 33

```
function f(x,y:T1) return T2;                      34
--| where return g(@x, @y);
```

But we need the identifier in cases where we only wish to provide a partial specification. 35

36
```
function f(x,y:T1) return T2;
--| where return z, z < g(@x, @y);
```

37 In the following example we require that the value of x be less than the value of y when f is called.

38
```
function f(x,y:T1) return T2
--| where in(@x < @y);
```

4. Names and Expressions

The rules applicable to the different forms of name and expression, and to their evaluation, are given in this section. 1

4.1 Names

Names can denote declared entities, whether declared explicitly or implicitly (see 3.1). Names can also denote ♦ the results of type_conversions or function_calls; and subcomponents ♦ of objects and values♦. Finally, names can denote attributes of any of the foregoing. 1

Syntax

```
name ::=                                                2
    direct_name           | ♦
  | indexed_component     | ♦
  | selected_component    | attribute_reference
  | type_conversion       | function_call
  | character_literal

direct_name ::= identifier | ♦                          3

prefix ::= name | ♦                                     4
```

♦

Certain forms of name (indexed_components, selected_components, ♦ and attributes) include a prefix that is itself a name that denotes some related entity♦. 7

Abstract Syntax

8

apply ∈ Apply	== **apply** *expr apl*	Before overload resolution.
dot ∈ Dot	== **dot** *name sym*	
name ∈ Name	== *id* | *apply* | *dot*	
name ∈ Name	== *id* | *indexed* | *selected*	After overload resolution.

♦

Dynamic Semantics

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. 11

The evaluation of a name that has a prefix includes the evaluation of the prefix. The evaluation of a prefix consists of the evaluation of the name ♦. The prefix denotes the entity denoted by the name ♦. 12

♦ 13

Examples

Examples of direct names: 14

15
♦
Limit -- *the direct name of a constant* (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)
Increment -- *the direct name of a function* (see 6.1)
♦

♦

4.1.1 Indexed Components

1 An indexed_component denotes a component of an array ♦.

Syntax

2 indexed_component ::= prefix(expression {, expression})

Abstract Syntax

3 *indexed* ∈ IndexedComponent == **indexed** *expr expr*

Name Resolution Rules

4 The prefix of an indexed_component with a given number of expressions shall resolve to denote an array ♦ with the corresponding number of index positions♦.

5 The expected type for each expression is the corresponding index type.

Static Semantics

6 ♦ The indexed_component denotes the component of the array with the specified index value(s). The nominal subtype of the indexed_component is the component subtype of the array type.

7 ♦

Dynamic Semantics

8 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. A check is made that each index value belongs to the corresponding index range of the array ♦ denoted by the prefix. Constraint_Error is raised if this check fails.

Examples

9 *Examples of indexed components:*

10
```
Filter(1)        --  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)
```
♦

NOTES

11 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). ♦

4.1.2 Slices -- Removed

4.1.3 Selected Components

Selected_components are used to denote components ♦; they are also used as expanded names as described below. 1

Syntax

selected_component ::= prefix . selector_name 2

selector_name ::= identifier | ♦ 3

Abstract Syntax

4

selected ∈ SelectedComponent == **selected** *expr sym*

Name Resolution Rules

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♦. 5

A selected_component that is not an expanded name shall resolve to denote one of the following: ♦ 6

- A component♦: 7

  The prefix shall resolve to denote an object or value of some record type ♦. The selector_name shall resolve to denote a ♦ component_declaration of the type. The selected_component denotes the corresponding component of the object or value. ♦ 8

- ♦ 9

  ♦ 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 a package ♦. 12
- The selector_name shall resolve to denote a declaration that occurs immediately within the declarative region of the package ♦ (the declaration shall be visible at the place of the expanded name — see 8.3). The expanded name denotes that declaration. 13
- ♦ 14

Dynamic Semantics

The evaluation of a selected_component includes the evaluation of the prefix. 15

♦ 16

Examples

Examples of selected components: 17

18

```
  Tomorrow.Month        -- a record component              (see 3.8)
  ♦
  Next_Person.Vehicle_Number   -- a record component      (see 3.8)
♦
```

19 *Examples of expanded names:*

20
```
Table_Manager.Insert -- a procedure of the visible part of a package (see 7.3)
♦
Standard.Boolean     -- the name of a predefined type                (see A.1)
```

4.1.4 Attributes

1 An *attribute* is a characteristic of an entity that can be queried via an attribute_reference or a range_ attribute_reference.

Syntax

2 attribute_reference ::= prefix'attribute_designator

3 attribute_designator ::=
   identifier[(*static*_expression)]
   | ♦

4 range_attribute_reference ::= prefix'range_attribute_designator

5 range_attribute_designator ::= Range[(*static*_expression)]

Abstract Syntax

6
attr ∈ Attr == **attr** *id sym* [*expr*]

Name Resolution Rules

7 ♦

8 The expression, if any, in an attribute_designator or range_attribute_designator is expected to be of any integer type.

Legality Rules

9 The expression, if any, in an attribute_designator or range_attribute_designator shall be static.

Static Semantics

10 An attribute_reference denotes a value, an object, a subprogram, or some other kind of program entity.

11 A range_attribute_reference X'Range(N) is equivalent to the range X'First(N) .. X'Last(N)♦. Similarly, X'Range is equivalent to X'First .. X'Last♦

Dynamic Semantics

12 The evaluation of an attribute_reference (or range_attribute_reference) has an effect that depends on the specific attribute. The result of this evaluation may be a value or a type.

Implementation Permissions

13 An implementation may provide implementation-defined attributes; the identifier for an implementation-defined attribute shall differ from those of the language-defined attributes.

NOTES

14 4 Attributes are defined throughout this Reference Manual, and are summarized in Annex J.

15 5 In general, the name in a prefix of an attribute_reference (or a range_attribute_reference) has to be resolved without using any context. ♦

Examples

Examples of attributes: 16

17
```
Color'First          -- minimum value of the enumeration type Color   (see 3.5.1)
Rainbow'Base'First   -- same as Color'First                           (see 3.5.1)
♦
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)
♦
```

4.2 Literals

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, ♦ or a string_literal. 1

Name Resolution Rules

♦ 2

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

The expected type for a primary that is a string_literal shall be a single string type. 4

Legality Rules

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. 5

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. 6

♦ 7

Abstract Syntax

8

$literal_a \in$ ArrayLiteral == $(n \,.\, expr)^*$
$literal_r \in$ RecordLiteral == $(sym \,.\, expr)^*$

Static Semantics

An integer literal is of type *universal_integer*. ♦ 9

Dynamic Semantics

♦ 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. 9

10 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. The exception Constraint_Error is raised if either of these checks fails.

NOTES

11 6 Enumeration literals that are identifiers rather than character_literals follow the normal rules for identifiers when used in a name ♦

Examples

12 *Examples of literals:*

13

```
♦
1_345         -- an integer literal
'A'           -- a character literal
"Some Text"   -- a string literal
```

4.3 Aggregates

1 An *aggregate* combines component values into a composite value of an array type or record type♦.

Syntax

2 aggregate ::= record_aggregate | ♦ | array_aggregate

Name Resolution Rules

3 The expected type for an aggregate shall be a single ♦ array type or record type♦.

Legality Rules

4 ♦

Dynamic Semantics

5 For the evaluation of an aggregate, an anonymous object is created and values for the components ♦ are obtained (as described in the subsequent subclause for each kind of the aggregate) and assigned into the corresponding components ♦ of the anonymous object. Obtaining the values and the assignments occur in an arbitrary order . The value of the aggregate is the value of this object. ♦

6 ♦

♦

4.3.1 Record Aggregates

1 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.

Syntax

2 record_aggregate ::= (record_component_association_list)

3 record_component_association_list ::=
    record_component_association {, record_component_association}
  | ♦

record_component_association ::= 4
  [component_choice_list =>] expression

component_choice_list ::= 5
  *component_*selector_name {| *component_*selector_name}
  | **others**

A record_component_association is a *named component association* if it has a component_choice_list; otherwise, it is a *positional component association*. ♦ Named and positional component associations cannot be used in the same aggregate. 6

In the record_component_association_list for a record_aggregate, if there is only one association, it shall be a named association. ♦ 7

Abstract Syntax

8

choice ∈ Choice	== *range* | *expr* | **others**
choices ∈ Choices	== **choices** *choice**
agg-choice ∈ AggChoice	== **agg-choice** *choices expr*
agg-pos ∈ AggPos	== **agg-pos** *expr*
agg-arm ∈ AggArm	== *agg-choice* | *agg-pos*
aggregate ∈ Aggregate	== **aggregate** *agg-arm**

Name Resolution Rules

The expected type for a record_aggregate shall be a single ♦ record type ♦. 9

For the record_component_association_list of a record_aggregate, all components of the composite value defined by the aggregate are *needed*♦. Each selector_name in a record_component_association shall denote a needed component ♦. 10

The expected type for the expression of a record_component_association is the type of the *associated* component(s); the associated component(s) are as follows: 11

- For a positional association, the component ♦ in the corresponding relative position (in the declarative region of the type), counting only the needed components;♦ 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 ♦. 14

Legality Rules

♦ 15

Each record_component_association 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 has two or more associated components, all of them shall be of the same type. 16

♦ 17

Dynamic Semantics

The evaluation of a record_aggregate consists of the evaluation of the record_component_association_list. 18

19 ♦ Any ♦ expression evaluations (and conversions) occur in an arbitrary order ♦.

20 The expression of a record_component_association is evaluated (and converted) once for each associated component.

♦

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 ♦

27 *Example of component association with several choices:*

28
```
    ♦
    (Month => July, Day|Year => 0)   -- see 3.8
```

29 ♦

4.3.2 Extension Aggregates -- Removed

4.3.3 Array Aggregates

1 In an array_aggregate, values are specified for each component of an array, either positionally or by ♦the choice **others**. For a positional_array_aggregate, the components are given in increasing-index order♦. For a named_array_aggregate, the components are identified by the values covered by the discrete_choices.

♦

Syntax

2
```
    array_aggregate ::=
      positional_array_aggregate | named_array_aggregate
```

3
```
    positional_array_aggregate ::=
        (expression, expression {, expression})
      | ♦
```

4
```
    named_array_aggregate ::=
       (others => expression)
```

5 ♦

6 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). 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. 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. ♦

Name Resolution Rules

The expected type for an array_aggregate (that is not a subaggregate) shall be a single ♦ array type. The component type of this array type is the expected type for each array component expression of the array_aggregate. 7

♦ 8

Legality Rules

An array_aggregate of an n-dimensional array type shall be written as an n-dimensional array_aggregate. 9

An **others** choice is allowed for an array_aggregate only if an *applicable index constraint* applies to the array_aggregate. 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: 10

- For an explicit_actual_parameter, ♦ the expression of a return_statement, or the initialization expression in an object_declaration, ♦ when the nominal subtype of the corresponding formal parameter, ♦ function result, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype. 11
- 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
- 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; 13
- 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
- For a parenthesized expression, the applicable index constraint is that, if any, defined for the expression. 15

The applicable index constraint *applies* to an array_aggregate that appears in such a context, as well as to any subaggregates thereof. ♦ 16

♦ 17

♦ 18

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. 19

Static Semantics

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. 20

Dynamic Semantics

The evaluation of an array_aggregate of a given array type proceeds in one step. 21

1. 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. 22

23 The bounds of the index range of an array_aggregate (including a subaggregate) are determined as follows:

24
- For an array_aggregate with an **others** choice, the bounds are those of the corresponding index range from the applicable index constraint;

25
- For a positional_array_aggregate (or equivalent string_literal) ♦, 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);

26
- ♦

27 For an array_aggregate, a check is made that the index range defined by its bounds is compatible with the corresponding index subtype.

28 ♦

29 For a multidimensional array_aggregate, a check is made that all subaggregates that correspond to the same index have the same bounds.

30 The exception Constraint_Error is raised if any of the above checks fail.

NOTES

31 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 (**others** => X), may be used to specify an array with a single component.

Examples

32 *Examples of array aggregates with positional associations:*

33
```
(7, 9, 5, 1, 3, 2, 4, 8, 6, 0)
♦
```

34

38 *Examples of two-dimensional array aggregates:*

39
```
♦
((1, 1, 1), (2, 2, 2))

(others => (1, 1, 1))
```

41 *Examples of aggregates as initial values:*

42
```
A : Table := (7, 9, 5, 1, 3, 2, 4, 8, 6, 0);          -- A(1)=7, A(10)=0
B : Table := (♦ others => 0);         -- B(i)=0 for i in 1..10
♦

♦
E : Bit_Vector(M .. N) := (others => True);
F : String(1 .. 1) := (♦ others => 'F');  -- a one component aggregate: same as "F"
```

4.4 Expressions

1 An *expression* is a formula that defines the computation or retrieval of a value. In this Reference Manual, the term ‘‘expression’’ refers to a construct of the syntactic category expression or of any of the other five syntactic categories defined below.

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 | ♦ | string_literal | aggregate
 | name | qualified_expression | ♦ | (expression)
```

Name Resolution Rules

A name used as a primary shall resolve to denote ♦ a value. 8

Abstract Syntax

9

expr ∈ Expr == *literal* | *expr* O_2 *expr* | O_1 *expr* | *aggregate* | *name* | *convert* | *qualified* | *function-call*>

Static Semantics

Each expression has a type; it specifies the computation or retrieval of a value of that type. 10

Dynamic Semantics

The value of a primary that is a name denoting an object is the value of the object. 11

Implementation Permissions

For the evaluation of a primary that is a name denoting an object of an unconstrained numeric subtype[3], if the value of the object is outside the base range of its type, the implementation must ♦ raise Constraint_ Error. ♦ 12

Examples

Examples of primaries: 13

```
♦                                                                       14
Pi                    -- named number
   (1, 2, 3, 4, 5)    -- array aggregate
Sum                   -- variable
Integer'Last          -- attribute
Abs(X)                -- function call
Color'(Blue)          -- qualified expression
♦
(Line_Count + 10)     -- parenthesized expression
```

Examples of expressions: 15

[3]The only unconstrained numeric subtype permitted in AVA is integer'Base.

16
```
Volume                                                              -- primary
not Destroyed            -- factor
2*Line_Count                                                        -- term
-4                       -- simple expression
-4 + A                   -- simple expression
B**2 - 4*A*C             -- simple expression
♦
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)
```

4.5 Operators and Expression Evaluation

1 The language defines the following six categories of operators (given in order of increasing precedence). ♦

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**

Abstract Syntax

8 And then and or else forms have been normalized to if_expressions.

$O_2 \in$ BinaryOperators	$== O_l \mid O_= \mid O_+ \mid O_*$
$O_\wedge$	== **and** | **or** | **xor**
$O_=$	== **in** | **not-in** | **isin** | **not-isin** | **=** | **ne** | **lt** | **gt** | **le** | **ge**
O_+	== **+** | **-** | **&**
O_*	== ***** | **/** | **mod** | **rem**
$O_1 \in$ UnaryOperators	== **abs** | **minus**

Static Semantics

9 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.

10 For each form of type definition, certain of the above operators are *predefined*; that is, they are implicitly declared immediately after the type definition. 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. ♦ The predefined operators and their effects are described in subclauses 4.5.1 through 4.5.6.

Dynamic Semantics

11 The predefined operations on integer types either yield the mathematically correct result or raise the exception Constraint_Error. ♦ ♦

Implementation Requirements

12 The implementation of a predefined operator that delivers a result of an integer ♦ type may raise Constraint_Error only if the result is outside the base range of the result type.

♦ 13

♦

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). Note that nothing in this section permits optimization of (1 + System.Max_Int - 1) to System.Max_Int. 14

Examples

Examples of precedence: 15

```
not Sunny or Warm      -- same as (not Sunny) or Warm
     X > 4 and Y > 0   -- same as   (X > 4) and (Y > 0)

4*A**2              -- same as –(4 * (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
```
16

4.5.1 Logical Operators and Short-circuit Control Forms

Name Resolution Rules

1 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; the expected type for both relations is that same boolean type.

Static Semantics

2 The following logical operators are predefined for every boolean type *T* ♦ and for every one-dimensional array type *T* whose component type is a boolean type[4]:

3
```
function "and"(Left, Right : T) return T
function "or" (Left, Right : T) return T
function "xor"(Left, Right : T) return T
```

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

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 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 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. Also, a check is made that each component of the result belongs to the component subtype. The exception Constraint_Error is raised if either of the above checks fails.

[4]We had intended to delete these overloadings. Unfortunately that would transform some *ambiguous* Ada programs into *unambiguous* AVA programs.

9 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
♦
```

13 *Examples of short-circuit control forms:*

14
```
♦
Next_Person.Age /= 0 and then 25 / Next_Person.Age < 1   -- see 3.8}
N = 0 or else A(N) = Hit_Value
```

4.5.2 Relational Operators and Membership Tests

1 The *equality operators* = (equals) and /= (not equals) are predefined for all types. The other relational_ operators are the *ordering operators* < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). 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. ♦

2 A *membership test*, using **in** or **not in**, determines whether or not a value belongs to a given subtype or range♦. Membership tests are allowed for all types.[5]

Name Resolution Rules

3 The *tested type* of a membership test is the type of the range or the type determined by the subtype_mark. ♦ The expected type for the simple_expression is the tested 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* ♦ with the following specifications:

7
```
function "=" (Left, Right : T) return Boolean
function "/="(Left, Right : T) 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
```

[5]They are *operations*, not operators or functions [AI-00128]. Presumably this is what allows then to take types as arguments.

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

♦ 11

For a private type, ♦ predefined equality for the private type is that of its full type. 15

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: 16

- For two composite objects or values of the same non-array type, matching components are those that correspond to the same component_declaration ♦; 17
- 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; 18
- For two multidimensional arrays of the same type, matching components are those whose index values match in successive index positions. 19

The analogous definitions apply if the types of the two objects or values are convertible, rather than being the same. 20

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: 21

- If there are no components, the result is defined to be True; 22
- If there are unmatched components, the result is defined to be False; 23
- Otherwise, the result is defined in terms of ♦ the predefined equals for any matching ♦ components. ♦ 24

The predefined "/=" operator gives the complementary result to the predefined "=" operator. 25

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

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
- The tested type is not scalar, and the value of the simple_expression satisfies any constraints of the named subtype ♦. 30

31 Otherwise the test yields the result False.

32 A membership test using **not in** gives the complementary result to the corresponding membership test using **in**.

NOTES

33 12 No exception is ever raised by a membership test, by a predefined ordering operator, or by a predefined equality operator for an elementary type, but an exception can be raised by the evaluation of the operands. ♦

34 13 ♦

Examples

35 *Examples of expressions involving relational operators and membership tests:*

36
```
X /= Y

"" < "A" and "A" < "Aa"      -- True
"Aa" < "B" and "A" < "A  "   -- True

♦

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)
♦
```

♦

4.5.3 Binary Adding Operators

Static Semantics

1 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 The concatenation operators & are predefined for every ♦ 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 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 ♦ array type was defined by a constrained_array_definition, then the lower bound of the result is that of the index subtype;

7 • If 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. 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 is raised if this check fails.

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. 9

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

♦

Examples

Examples of expressions involving binary adding operators: 12

♦ 13

```
"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
```

4.5.4 Unary Adding Operators

Static Semantics

The unary adding operators + (identity) and – (negation) are predefined for every specific numeric type *T* with their conventional meaning. They have the following specifications: 1

2
```
function "+"(Right : T) return T
function "-"(Right : T) return T
```

♦

4.5.5 Multiplying Operators

Static Semantics

The multiplying operators * (multiplication), / (division), **mod** (modulus), and **rem** (remainder) are predefined for every specific integer type *T*: 1

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
```

Signed integer multiplication has its conventional meaning. 3

Signed integer division and remainder are defined by the relation: 4

5
```
A = (A/B)*B + (A rem B)
```

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

7
```
(-A)/B = -(A/B) = A/(-B)
```

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

9
```
A = B*N + (A mod B)
```

10 ♦

Dynamic Semantics

21 ♦

22 The exception Constraint_Error is raised by integer division, **rem**, and **mod** if the right operand is zero. ♦

NOTES

23 17 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 18 For any signed integer K, the following identity holds (ignoring exceptions):

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

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

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;
```

♦

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*

♦

♦

4.5.6 Highest Precedence Operators

Static Semantics

1 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
```

The highest precedence unary operator **not** (logical negation) is predefined for every boolean type *T*, ♦ and for every one-dimensional array type *T* whose components are of a boolean type, with the following specification: 3

```
    function "not"(Right : T) return T
```
4

♦ 5

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). 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

The highest precedence *exponentiation* operator ** is predefined for every specific integer type *T* with the following specification: 7

```
    function "**"(Left : T; Right : Natural) return T
```
8

♦ 9

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. ♦ 11

♦

NOTES

19 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 is raised if this check fails. 13

4.6 Type Conversions

Explicit type conversions ♦ are allowed between closely related types as defined below. This clause also defines rules for value ♦ conversions to a particular subtype of a type, both explicit ones and those implicit in other constructs. 1

Syntax

```
type_conversion ::=
   subtype_mark(expression)
  | ♦
```
2

The *target subtype* of a type_conversion is the subtype denoted by the subtype_mark. The *operand* of a type_conversion is the expression ♦ within the parentheses; its type is the *operand type*. 3

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

♦ ♦ Type_conversions are called *value conversions*. 5

Abstract Syntax

6 *convert* ∈ TypeConvert == **convert** *type expr*

Name Resolution Rules

7 The operand of a type_conversion is expected to be of any type.

8 ♦ The operand of a value conversion is interpreted as an expression.

Legality Rules

9 If the target type is a numeric type, then the operand type shall be a numeric type.

10 If the target type is an array type, then the operand type shall be an array type. Further:

- 11 The types shall have the same dimensionality;
- 12 Corresponding index types shall be convertible; and
- 13 The component subtypes shall statically match.

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 ♦

27 The nominal subtype of a type_conversion is its target subtype.

Dynamic Semantics

28 For the evaluation of a type_conversion ♦ the operand is evaluated, and then the value of the operand is *converted* to a *corresponding* value of the target type, if any. If there is no value of the target type that corresponds to the operand value, Constraint_Error is raised♦. Additional rules follow:

- 29 Numeric Type Conversion
  - 30 ♦ The result is the value of the target type that corresponds to the same mathematical integer as the operand.
  - 31 ♦
  - 32 ♦
  - 33 ♦
- 34 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 Array Type Conversion
  - 37 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 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. For each nonnull index range, a check is made that the bounds of the range belong to the corresponding index subtype.

  - 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

- ♦ 40
- ♦ 41

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. 51

♦ 52

♦ 53

♦ Any ♦ 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

NOTES

20 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. ♦ 59

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. 60

21 A ramification of the overload resolution rules is that the operand of an (explicit) type_conversion cannot be ♦ an aggregate, a string_literal, or a character_literal♦. 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. 61

22 ♦ 62

Examples

♦ 63

Examples of conversions between array types: 69

```
type Sequence is array (Integer range <>) of Integer;                70
subtype Dozen is Sequence(1 .. 12);
Ledger : array Integer range (1 .. 100) of Integer;

Sequence(Ledger)            -- bounds are those of Ledger
♦
```

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. 1

Syntax

```
qualified_expression ::=                                             2
  subtype_mark'(expression) | subtype_mark'aggregate
```

Name Resolution Rules

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. 3

Abstract Syntax

4 *qualified* ∈ Qualified == **qualified** *type expr*

Dynamic Semantics

5 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. The exception Constraint_Error is raised if this check fails.

NOTES

6 23 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

7 *Examples of disambiguating expressions using qualification:*

8
```
type Mask is (Fix, Dec, Exp, Signif);
type Code is (Fix, Cla, Dec, Tnz, Sub);

Print (Mask'(Dec));   --  Dec is of type Mask
Print (Code'(Dec));   --  Dec is of type Code

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
```

♦

4.8 Allocators -- Removed

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* means determinable at compile time, using the declared properties or values of the program entities.

2 A *static expression* is a a scalar or string expression that is one of the following:

3 • a numeric_literal;

4 • ♦

5 • a name that denotes the declaration of a named number or a static constant;

6 • 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;

7 • an attribute_reference that denotes a scalar value, and whose prefix denotes a static scalar subtype;

8 • ♦

9 • ♦

10 • a qualified_expression whose subtype_mark denotes a static (scalar or string) subtype, and whose operand is a static expression;

- ♦ 11
- ♦ 12
- a static expression enclosed in parentheses. 13

14 A name *statically denotes* an entity if it denotes the entity and:

- 15 It is a direct_name, expanded name, or character_literal, and it denotes a declaration other than a renaming_declaration; or
- 16 It is an attribute_reference whose prefix statically denotes some entity; or
- 17 It denotes a renaming_declaration with a name that statically denotes the renamed entity.

18 A *static function* is one of the following:

- 19 a predefined operator whose parameter and result types are all scalar types ♦;
- 20 a predefined concatenation operator ♦;
- 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 ♦

24 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 ♦. ♦

25 A *static range* is a range whose bounds are static expressions, or a range_attribute_reference that is equivalent to such a 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 ♦.

26 A *static subtype* is ♦ a *static scalar subtype* ♦. A static scalar subtype is an unconstrained scalar subtype ♦ or a constrained scalar subtype formed by imposing a compatible static constraint on a static scalar subtype. ♦

27 The different kinds of *static constraint* are defined as follows:

- 28 A null constraint is always static;
- 29 A scalar constraint is static if it has no range_constraint, or one with a static range;
- 30 An index constraint is static if each discrete_range is static, and each index subtype of the corresponding array type is static;
- 31 ♦

32 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 • If the expression is not part of a larger static expression, then its value shall be within the base range of its expected type. Otherwise, the value may be arbitrarily large or small.

36 • 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.

37 The last two restrictions above do not apply if the expected type is a descendant of a formal scalar type (or a corresponding actual type in an instance).

36 ♦

♦

NOTES

36 28 An expression can be static even if it occurs in a context where staticness is not required.

37 29 ♦

Examples

41 *Examples of static expressions:*

42
```
1 + 1        -- 2
abs(-10)*3   -- 30

Kilo : constant := 1000;
Mega : constant := Kilo*Kilo;    -- 1_000_000
♦
```

FORMAL NOTES

42 We have allowed the semantics of static expressions to remain unchanged from Ada95. We were previously concerned about having two different models of arithmetic to worry about. But that problem is present anyway, given mathematical integers and Integers. It is still unclear whether this approach will work, or whether we really end up with three models.

4.9.1 Statically Matching Constraints and Subtypes

Static Semantics

1 A constraint *statically matches* another constraint if both are null constraints, both are static and have equal corresponding bounds ♦, or 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.

2 A subtype *statically matches* another subtype of the same type if they have statically matching constraints. ♦

3 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. ♦

4 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. A constraint is *statically compatible* with ♦ a composite subtype if it statically matches the constraint of the subtype, or if the subtype is unconstrained. One subtype is *statically compatible* with a second subtype if the constraint of the first is statically compatible with the second subtype.

4.10 Logical Expressions

A *logical expression* is a formula that defines a value in the ACL2 logic. The syntax of *logical*_expression is simply an extension of expression syntax to annotation contexts. While *logical*_expression syntax includes expression, the interpretation of expressions is quite different within a logical context. For example there is no ambiguity between function_call and indexed_component. Expressions of the form ‘‘name(arg$_1$, ...)’’ are always function or macro calls within the ACL2 logic. Logical contexts permit functions of no arguments. 1

2

Syntax

3
```
logical_expression ::=
    expression
  | env_expression
  | if logical_expression
      then expression
      else expression
    fi
  | logical_expression iff logical_expression
  | logical_expression implies logical_expression
  | all identifier [ in logical_expression ] , logical_expression
```

4
```
env_expression ::=
    @ identifier
  | in expression
  | out expression
```

Abstract Syntax

5

list ∈ List	== *lexpr**	
LLiteral ∈ LLiteral	== *sym* | *literal* | **t** | **nil** | *list*	
LO_2 ∈ BinaryLOperators	== LO_l | $LO_=$ | LO_+ | LO_* | LO_{op}	
$LO_\wedge$	== **in** | **not-in** | **isin** | **not-isin** | **iff** | -> | **and** | **or**	
$LO_=$	== = | **..** | **ne** | **lt** | **gt** | **le** | **ge**	
LO_+	== + | - | **&**	
LO_*	== ***** | / | **mod** | **rem**	
LO_{op}	== **append** | **power**	
LO_1 ∈ UnaryOperators	== **abs** | **minus** | **not**	
function-call ∈ FunctionCall	== *sym lexpr**	
instate ∈ InState	== **instate** *lexpr*	
outstate ∈ OutState	== **outstate** *lexpr*	
forall ∈ Forall	== **forall** *id lexpr lexpr*	
if ∈ If	== **if** *lexpr lexpr lexpr*	
set ∈ Set	== **set** *lexpr lexpr lexpr*	E.g. foo[i := 10]
get ∈ Get	== **get** *lexpr lexpr*	E.g. foo[10]
assoc ∈ Assoc	== **assoc** *lexpr lexpr*	
lookup ∈ Lookup	== **lookup** *sym vs*	
in-range ∈ InRange	== **in-range** *lexpr lexpr*	
lexpr ∈ LExpr	== *LLiteral* | *lexpr* LO_2 *lexpr* | LO_1 *lexpr* *forall* | *if* | *get* | *set* | *assoc* | *lookup* | *in-range* | *function-call* | *instate outstate*	

Name Resolution Rules

6 The names in logical_expressions must resolve to logical variables, functions, or macros unless they are prefixed by **@**, in which case they must reference program identifiers.

Formal Dynamic Semantics

7 The semantics of expressions contained in logical expressions are different from the normal Ada semantics. See the table below for the tranlation from infix operators to the corresponding ACL2 function names. The operational meaning of these expressions depends on the annotation context in which they appear.

8 The ‘‘@’’ operator provides the fundamental communication between the Ada state and ACL2. An Ada variable prefixed by @, say @x, is interpreted to mean the value of the variable in the state, which defaults to the *current* state. A state is actually a tuple consisting of a value stack (for function evaluation), an entity stack (containing mappings from unique identifiers to subprogram bodies and objects, including their type and value), and an assertion stack (to hold the logical forms that must evaluate to **true** in the current context).

9 The **in** operator establishes an environment within which the state is the initial state, as defined by the surrounding logical context. The **out** operator establishes an environment within which the state is the current state. Thus a subprogram annotion of the form ‘‘**in** @x + @y = **out** @x + @y’’ asserts that the sum of two variables is identical in the initial state (before the execution of the procedure body) and the current state (after the execution of the procedure body). This could be stated without using **out**, but it is sometimes clearer.

10 **If then else** provides the functionality of the ACL2 ‘‘cond’’.

11

Operator Translations	
Operator	**ACL2 Function**
implies, **iff**	implies, iff
and, **or**	and, or
in	member-equal
=	equal
&	append
+, **-**	+, **-**
*, /, **mod**, **rem**	*, /, mod, rem
, **abs, **not**	expt, abs, not
[a, b, c]	(list a b c)
x[i := 1]	(set-t x i 1)
x[i]	(get-t x i)

5. Statements

A statement defines an action to be performed upon its execution. 1

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 Section 6, ‘‘Subprograms’’. ♦ Raise_statements are described in Section 11, ‘‘Exceptions’’♦. The remaining forms of statements are presented in this section. 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
  ♦ simple_statement | ♦ ava_compound_statement

simple_statement ::=                                                 4
    null_statement            | assert_annotation
  | assignment_statement      | exit_statement
  | ♦                         | procedure_call_statement
  | return_statement          | ♦
  | ♦                         | ♦
  | ♦                         | raise_statement
  | ♦

compound_statement ::=                                               5
    if_statement              | case_statement
  | loop_statement            | block_statement

ava_compound_statement ::= compound_statement [ transitin_annotation ]   6

null_statement ::= null;                                             7

♦                                                                    8

♦                                                                    9

♦                                                                    14
```

Abstract Syntax

15

$null \in$ Null == **null**

Dynamic Semantics

The execution of a null_statement has no effect. 13

A *transfer of control* is the run-time action of an exit_statement or return_statement♦ or the raising of an exception, ♦ which causes the next action performed to be one other than what would normally be expected from the other rules of the language. 14

The execution of a sequence_of_statements consists of the execution of the individual statements in succession until the sequence_ is completed. The elaboration of a transition_annotation of an ava_ 15 compound_statement asserts it as an *invariant* throughout the execution of the compound_statement. An invariant must be true after every simple statement within the scope of the compound_statement.

Dynamic Semantics

16 .

♦

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*. An assignment operation (as opposed to an assignment_ statement) is performed in other contexts as well, including object initialization and by-copy parameter passing. 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.

Abstract Syntax

4

assign ∈ Assign == **assign** *name expr*

Name Resolution Rules

5 The *variable*_name of an assignment_statement is expected to be of any ♦ type. The expected type for the expression is the type of the target.

Legality Rules

6 The target denoted by the *variable*_name shall be a variable.

7 ♦

Dynamic Semantics

8 For the execution of an assignment_statement, the *variable*_name and the expression are first evaluated in an arbitrary order.

9 ♦

11 The value of the expression is converted to the subtype of the target. The conversion might raise an exception (see 4.6).

12 ♦ ♦ The converted value of the expression is then *assigned* to the target♦.

♦

Examples

Examples of assignment statements: 17

```
Value := Max_Value - 1;                                        18
Shade := Blue;

♦                                                              19
U := Dot_Product(V, W);            -- see 6.3

♦                                                              20
Birthdate := (Day => 1, Month => May, Year => 1960);    -- see 3.8
```

Examples involving scalar subtype conversions: 21

```
I, J : Integer range 1 .. 10 := 5;                             22
K    : Integer range 1 .. 20 := 15;
 ...

I := J;   -- identical ranges                                  23
K := J;   -- compatible ranges
J := K;   -- will raise Constraint_Error if K > 10
```

Examples involving array subtype conversions: 24

```
♦                                                              25
subtype low_index  is integer range 1..20;
subtype high_index is integer range 3..22;
subtype S1 is STRING(low_index);
subtype S2 is STRING(high_index);

A : S1 := "This is a test.     ";                              26
B : S2 := A;                    --  same number of components
```

NOTES
1 ♦ 28

♦

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 A condition is expected to be of ♦ boolean type.

Abstract Syntax

5

ifarm ∈ IfArm	== **ifarm** *expr stmt**
if-stmt ∈ IfStmt	== **if-stmt** *ifarm**

Dynamic Semantics

6 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.

Examples

7 *Examples of if statements:*

8
```
if Month = December and Day = 31 then
   Month := January;
   Day   := 1;
   Year  := Year + 1;
end if;
```

9
```
if Line_Too_Short then
   raise Program_Error;
elsif Line_Full then
   Put(Ofile, EOL);
   Put(Ofile, Item);
else
   Put(Ofile, Item);
end if;
```

10
```
♦
if Next_Person.Vehicle.Owner /= Next_Person.Name then          -- see 3.8
  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.[6]

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
```

3
```
discrete_choice_list ::= discrete_choice {| discrete_choice}
```

3
```
discrete_choice ::= expression | discrete_range | others
```

Name Resolution Rules

4 The expression is expected to be of any discrete type. The expected type for each discrete_choice is the type of the expression.

[6]The grammar productions for discrete_choice_list and discrete_choice appeared in Section 3.8.1 of Ada95 Standard, but were moved here because that section has been removed from the AVA report.

Abstract Syntax

5

casearm ∈ CaseArm	== **casearm** *choices stmt**
case-stmt ∈ CaseStmt	== **case-stmt** *expr casearm**

Legality Rules

A discrete_choice is defined to *cover a value*[7] in the following cases: 4

- A discrete_choice that is an expression covers a value if the value equals the value of the expression converted to the expected type. 5
- A discrete_choice that is a *integer*_range covers all values (possibly none) that belong to the range. 6
- The discrete_choice **others** covers all values of its expected type that are not covered by previous discrete_choice_lists of the same construct.[8] 7

A discrete_choice_list covers a value if one of its discrete_choices covers the value. 4

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. 5

The possible values of the expression shall be covered as follows: 6

- 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
- If the type of the expression is *root_integer* or *universal_integer*♦, then the case_statement shall have an **others** discrete_choice. 8
- Otherwise, each value of the base range of the type of the expression shall be covered (either explicitly or by **others**). 9

Two distinct discrete_choices of a case_statement shall not cover the same value. 10

Dynamic Semantics

For the execution of a case_statement the expression is first evaluated. 11

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. 12

Otherwise (the value is not covered by any discrete_choice_list, perhaps due to being outside the base range), Constraint_Error is raised. 13

[7]This paragraph extracted from section 3.8.1 of the AARM.

[8]Note that subsets of Ada that are concerned with safety rule out the use of **others** in order that all possible choices will be explicitly covered in the case statement.

NOTES

14 2 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 Program_Error;
end case;
```

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 ::=
  ♦
   [iteration_scheme] **loop**
    sequence_of_statements
   end loop ♦;

3 iteration_scheme ::= **while** condition
  | **for** loop_parameter_specification

4 loop_parameter_specification ::=
  defining_identifier **in** [**reverse**] discrete_subtype_definition

5 ♦

Abstract Syntax

6

loop ∈ Loop	== **loop** *stmt**
while-loop ∈ WhileLoop	== **while-loop** *expr stmt**
for-loop ∈ ForLoop	== **for-loop** *id range stmt**
reverse-for-loop ∈ ReverseForLoop	== **reverse-for-loop** *id range stmt**
loop-stmt ∈ LoopStmt	== *loop* | *while-loop* | *for-loop* | *reverse-for-loop*

Static Semantics

A loop_parameter_specification declares a *loop parameter*, which is an object whose subtype is that defined by the discrete_subtype_definition. 7

Dynamic Semantics

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

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

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). 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. 10

NOTES

3 A loop parameter is a constant; it cannot be updated within the sequence_of_statements of the loop (see 3.3). 11

4 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. 12

5 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

14
```
for J in reverse 1 ..  0
for J in 0 .. 1
```

Examples

Example of a loop statement without an iteration scheme: 15

16
```
loop
   Get(Current_Character);
   exit when Current_Character = '*';
end loop;
```

Example of a loop statement with a **while** *iteration scheme:* 17

18
```
while Bid(N).Price < Cut_Off.Price loop
   Record_Bid(Bid(N).Price);
   N := N + 1;
end loop;
```

Example of a loop statement with a **for** *iteration scheme:* 19

20
```
for J in Buffer'Range loop     -- works even with a null range
   if Buffer(J) /= Space then
      Put(Standard_Output, Buffer(J));
   end if;
end loop;
```

21 ♦

5.6 Block Statements

1 A block_statement encloses a handled_sequence_of_statements optionally preceded by a declarative_part.

Syntax

2
```
block_statement ::=
  ♦
    [declare
       inner_part]
     begin
       handled_sequence_of_statements
     end ♦;
```

♦

Abstract Syntax

3 *block* ∈ Block == **block** [d_i] [*handler*] $stmt^*$

Static Semantics

4 A block_statement that has no explicit inner_part has an implicit empty inner_part.

Dynamic Semantics

5 The execution of a block_statement consists of the elaboration of its inner_part ollowed by the execution of its handled_sequence_of_statements.

Examples

6 *Example of a block statement with a local variable:*

7
```
♦
declare
   Temp : Integer := 1;
begin
   Temp := V; V := U; U := Temp;
end ♦;
```

♦

5.7 Exit Statements

1 An exit_statement is used to complete the execution of an enclosing loop_statement ♦.

Syntax

2
```
exit_statement ::=
  exit ♦ ;
```

♦

Abstract Syntax

3 *exit* ∈ Exit == **exit**

Legality Rules

4 Each exit_statement *applies to* a loop_statement; this is the loop_statement being exited. ♦ An exit_statement ♦ is only allowed within a loop_statement, and applies to the innermost enclosing one. ♦

Dynamic Semantics

5 For the execution of an exit_statement ♦ a transfer of control is done to complete the loop_statement. ♦

♦

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);
   if New_Item = Terminal_Item then @key[exit]; end if
end loop;
```

9 ♦

5.8 Goto Statements -- Removed

5.9 Assert Annotations -- New

The syntax and semantics of assert annotations is given in the sections on annotation declarations 3.12 and logical expressions 4.10.

6. Subprograms

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. One or both of the declaration and body can include a subprogram_annotation. The annotation of a declaration defaults to true. If a body is a completion of a declaration and the body includes a subprogram_annotation then the body annotation must imply the declaration annotation. The *primary* subprogram annotation of a subprogram is that of its body, if present, otherwise it is that of its declaration. 1

A *callable entity* is a subprogram ♦. A callable entity is invoked by a *call*; that is, a subprogram call ♦. A *callable construct* is a construct that defines the action of a call upon a callable entity: a subprogram_body♦. 2

6.1 Subprogram Declarations

A subprogram_declaration declares a procedure or function. 1

Syntax

```
subprogram_declaration ::=                                                  2
  subprogram_specification; [ subprogram_annotation; ]

♦                                                                           3

subprogram_specification ::=                                                4
    procedure defining_program_unit_name parameter_profile
  | function defining_designator  parameter_and_result_profile

designator ::= [parent_unit_name . ] identifier | ♦                         5

defining_designator ::= defining_program_unit_name | ♦                      6

defining_program_unit_name ::= [parent_unit_name . ] defining_identifier    7
```

The optional parent_unit_name is only allowed for library units (see 10.1.1). 8

```
♦                                                                           9

parameter_profile ::= [formal_part]                                         12

parameter_and_result_profile ::= [formal_part] return subtype_mark          13

formal_part ::=                                                             14
  (parameter_specification {; parameter_specification})

parameter_specification ::=                                                 15
   defining_identifier_list : mode  subtype_mark ♦
  | ♦

mode ::= [in] | in out | ♦                                                  16
```

Abstract Syntax

17

$fp \in$ FpSpec	== **fp** *id mode type*
$fpl \in$ Fpl	== **fpl** fp^*
$d_p \in$ Procedure	== **procedure** *id fpl* **nil** [*block*] [$spec_p$]

$d_f \in$ Function == **function** *id fpl id* [*block*] [$spec_p$]

subprogram $\in$ Subprogram == $d_f \mid d_p$

Name Resolution Rules

18 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. ♦

Legality Rules

19 The *parameter mode* of a formal parameter conveys the direction of information transfer with the actual parameter: **in** or **in out**♦. Mode **in** is the default♦. The formal parameters of a function♦ shall have the mode **in**. ♦

20 ♦

21 ♦ A subprogram_declaration ♦ requires a completion: a body ♦. ♦ ♦

22 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 ♦. ♦

Static Semantics

23 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 and other subprograms♦. ♦ Associated with a profile is a calling convention as well as a primary subprogram annotation. 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.

24 The nominal subtype of a formal parameter is the subtype denoted by the subtype_mark♦ in the parameter_specification.

25 ♦

26 The *subtypes of a profile* are:

27 • For any ♦ parameters, the nominal subtype of the parameter.

28 • ♦

29 • For any result, the result subtype.

30 The *types of a profile* are the types of those subtypes.

31 ♦

Dynamic Semantics

32 ♦ The elaboration of a subprogram_declaration ♦ has no effect.

NOTES

33 1 A parameter_specification with several identifiers is equivalent to a sequence of single parameter_specifications, as explained in 3.3.

34 2 ♦

3 ♦ 35

4 Subprograms can be called recursively ♦. 36

Examples

Examples of subprogram declarations: 37

```
procedure Traverse_Tree;                                    38
procedure Increment(X : in out Integer);
♦
procedure Switch(From, To : in out Color);
function Random(I : in Page_Num) return Page_Num;           39
♦                                                           40
function Birth_Date(K : Person) return Date;
```

♦ 41

6.2 Formal Parameter Modes

A parameter_specification declares a formal parameter of mode **in** or **in out**♦. 1

Static Semantics

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 body. ♦ 2

Bounded (Run-Time) Errors

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*. In Ada 95, objects assigned via one access path, and then read via a distinct access path may result in a bounded error. The AVA requirement for *by copy* semantics eliminates this bounded error. Note, however, that reasoning in such situations will be complicated. 3

NOTES

5 A formal parameter of mode **in** is a constant view (see 3.3); it cannot be updated within the subprogram_body. 13

♦

6.3 Subprogram Bodies

A subprogram_body specifies the execution of a subprogram. 1

Syntax

```
subprogram_body ::=                                          2
   subprogram_specification is
      inner_declarative_part
   begin
      handled_sequence_of_statements
   end [designator];
   [subprogram_annotation;]
```

If a designator appears at the end of a subprogram_body, it shall repeat the defining_designator of the subprogram_specification. 3

Legality Rules

4 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 ♦. The profile of a subprogram_body that completes a declaration shall conform fully to that of the declaration.

Static Semantics

5 A subprogram_body is considered a declaration. It can either complete a previous declaration, or itself be the initial declaration of the subprogram. A function subprogram_body is further restricted. It may not include any procedure calls or assignments to variables not local to some declarative region of the function body.

6 FORMAL NOTES

7 As a result of this restriction we can prove that for a list of expressions, l_1,

$$\mathrm{permutation}(l_1,l_2) \rightarrow \mathrm{permutation}(\mathrm{eval_l}(l_1), \mathrm{eval_l}(l_2))$$

That is, if l_2 is a permutation of l_1 then the result of evaluating the expressions in l_2 is a permutation of the result of evaluating the expressions in l_1. This property can be stated this simply because if $\mathrm{eval_l}$ raises any exception it will return an empty list.

Dynamic Semantics

6 The elaboration of a ♦ subprogram_body has no other effect than to establish that the subprogram can from then on be called without failing the Elaboration_Check. ♦

7 The execution of a subprogram_body is invoked by a subprogram call. For this execution the inner_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 Program_Error;
   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 Integer is
   Sum :    Integer := 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;
```

6.3.1 Conformance Rules

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. 1

Static Semantics

♦ A *convention* can be specified for an entity. For a callable entity ♦, the convention is called the *calling convention*. The following conventions are defined by the language: 2

- The default calling convention for any subprogram not listed below is *Ada*. ♦ 3
- 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
  - ♦ 6
  - any other implicitly declared subprogram ♦ 7
  - ♦ 8
  - an attribute that is a subprogram; 9
  - ♦ 10

  ♦ 11
- ♦ 12

♦ 13

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♦. ♦ 15

Two profiles are *mode conformant* if they are type-conformant, and corresponding parameters have identical modes♦. 16

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. ♦ 17

Two profiles are *fully conformant* if they are subtype-conformant, and corresponding parameters have the same names ♦. 18

Implementation Permissions

An implementation may declare an operator declared in a language-defined library unit to be intrinsic. 19

6.3.2 Inline Expansion of Subprograms -- Removed

6.4 Subprogram Calls

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

2 procedure_call_statement ::= *procedure*_**name** [actual_parameter_part] ;

3 function_call ::= ♦ | *function*_prefix actual_parameter_part

4 actual_parameter_part ::= (parameter_association {, parameter_association})

5 parameter_association ::= ♦ explicit_actual_parameter

6 explicit_actual_parameter ::= expression | *variable*_name

7 A parameter_association is ♦ *positional* ♦

Abstract Syntax

8

proc-call ∈ ProcCall	== **proc-call** *id expr**
function-call ∈ FunctionCall	== **function-call** *id expr**

Name Resolution Rules

9 The name ♦ given in a procedure_call_statement shall resolve to denote a callable entity that is a procedure♦. The ♦ prefix given in a function_call shall resolve to denote a callable entity that is a function. ♦

10 ♦ For each formal parameter of a subprogram, a subprogram call must specify exactly one corresponding actual parameter. This actual parameter is specified explicitly by a parameter association. The actual parameter corresponds to the formal parameter with the same position in the formal_part.

Dynamic Semantics

11 For the execution of a subprogram call, the name♦ of the call is evaluated, and each parameter_ association is evaluated (see 6.4.1). ♦ These evaluations are done in an arbitrary order. The current state is saved as the *initial* state. The subprogram_body is then executed. The primary subprogram annotation is evaluated against the inital state and the current state before control is passed back to the calling environment. 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).

12 The exception Program_Error is raised at the point of a function_call if the function completes normally without executing a return_statement.

13 A function_call denotes a constant, as defined in 6.5; the nominal subtype of the constant is given by the result subtype of the function.

12 It may be difficult to predict the behavior of programs in which the actual parameters corresponding to two different formal parameters of mode **in out** overlap.[9] Thus, given

13
```
    procedure Q(X, Y : in out T);
```
the call "Q(a[1], a[2])" is acceptable, but "Q(a[i], a[j])" will present problems for analysis.

[9]Two variables overlap if they are equal or either is a subcomponent of the other. Thus, the array **a** and **a**(1) overlap. Array elements **a**(i) and **a**(j) *potentially* overlap. Any interesting subprogram annotation will almost certainly require an anti-aliasing hypothesis.

Examples

Examples of procedure calls: 13

```
Traverse_Tree;                                       -- see 6.1    14
Table_Manager.Insert(E);
Print_Header(128, Title, True);                      -- see 6.1
```

♦ 15

Examples of function calls: 16

```
Dot_Product(U, V)   -- see 6.1 and 6.3    17
♦
```

♦ 18

Examples

Examples of overloaded subprograms: 25

```
procedure Put(X : in Integer);       26
procedure Put(X : in String);

procedure Set(Tint   : in Color);    27
procedure Set(Signal : in Light);
```

Examples of their calls: 28

```
Put(28);                                  29
Put("no possible ambiguity here");

♦                                         30
Set(Light'(Red));
Set(Color'(Red));

-- Set(Red) would be ambiguous since Red may          31
-- denote a value either of type Color or of type Light
```

6.4.1 Parameter Associations

A parameter association defines the association between an actual parameter and a formal parameter. 1

Name Resolution Rules

♦ 2

The *actual parameter* is ♦ the explicit_actual_parameter given in a parameter_association for a given formal parameter♦. The expected type for an actual parameter is the type of the corresponding formal parameter. ♦ 3

If the mode is **in**, the actual is interpreted as an expression; otherwise, the actual is interpreted only as a name, if possible. 4

Legality Rules

If the mode is **in out** ♦, the actual shall be a name that denotes a variable. Type conversions of actual parameters associated with an **in out** formal parameter are not allowed. ♦ ♦ 5

♦ 6

Dynamic Semantics

For the evaluation of a parameter_association: 7

- The actual parameter is first evaluated. 8

9 • ♦

10 • ♦

11 • ♦ 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.

12 • ♦

13 ♦

17 After normal completion and leaving of a subprogram, for all **in out** ♦ parameters♦, the value of the formal parameter is converted to the subtype of the variable given as the actual parameter and then the values are assigned to the respective variables[10] . ♦

6.5 Return Statements

1 A return_statement is used to complete the execution of the ♦ enclosing subprogram_body♦.

Syntax

2 return_statement ::= **return** [expression];

Abstract Syntax

3 *return* ∈ Return = **return** [*expr*]

Name Resolution Rules

4 The expression, if any, of a return_statement is called the *return expression*. The *result subtype* of a function is the subtype denoted by the subtype_mark after the reserved word **return** in the profile of the function. The expected type for a return expression is the result type of the corresponding function. ♦

Legality Rules

5 A return_statement shall be within a callable construct, and it *applies to* the innermost one. ♦

6 A function body shall contain at least one return_statement that applies to the function body♦. A return_statement shall include a return expression if and only if it applies to a function body.

Dynamic Semantics

7 For the execution of a return_statement, the expression (if any) is first evaluated and converted to the result subtype.

8 ♦

22 ♦A function result is returned by copy; that is, the converted value is assigned into an anonymous constant created at the point of the return_statement, and the function call denotes that object. ♦

23 Finally, a transfer of control is performed which completes the execution of the callable construct to which the return_statement applies, and returns to the caller.

[10] In Ada95, copy-back and constraint checking can be interleaved in an order that is not defined by the language.

Examples

Examples of return statements: 24

25
```
   return;                        -- in a procedure body♦
   return Key_Value(Last_Index);  -- in a function body
```

♦

6.6 Overloading of Operators -- Removed

7. Packages

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 ♦) 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. A package may also include axioms, purported theorems, and specification functions. 1

7.1 Package Specifications and Declarations

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. 1

Syntax

package_declaration ::= package_specification; 2

package_specification ::= 3
  package defining_program_unit_name **is**
    {basic_declarative_item}
  [**private**
    {basic_declarative_item}]
  end [[parent_unit_name.]identifier]

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. 4

Legality Rules

♦ A package_declaration ♦ requires a completion (a body) if it contains any declarative_item that requires a completion, but whose completion is not in its package_specification. ♦ 5

Abstract Syntax

6

outer	== d^*
private	== d^*
inner	== d^*
$p \in$ Package	== **package** *id* [*outer*] [*private*] [*inner*] [$stmt^*$]

Static Semantics

The first list of declarative_items of a package_specification of a package ♦ is called the *visible part* of the package. The optional list of 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. 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). 8

Dynamic Semantics

The elaboration of a package_declaration consists of the elaboration of its basic_declarative_items in the given order. 9

NOTES

10 1 The visible part of a package contains all the information that another program unit is able to know about the package.

11 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.

Examples

12 *Example of a package declaration:*

```
13  package Rational_Numbers is

14     type Rational is
          record
             Numerator   : Integer;
             Denominator : Positive;
          end record;

15     function Equal (X,Y : Rational) return Boolean;

16     function Div   (X,Y : Integer)  return Rational;  -- to construct a rational number

17     function Plus  (X,Y : Rational) return Rational;
       function Minus (X,Y : Rational) return Rational;
       function Times (X,Y : Rational) return Rational;
       function Div   (X,Y : Rational) return Rational;
    end Rational_Numbers;
```

18 There are also many examples of package declarations in the predefined language environment (see Annex A).

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

2
```
package_body ::=
   package body defining_program_unit_name is
      declarative_part
  [begin
       handled_sequence_of_statements]
   end [[parent_unit_name.]identifier];
```

3 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.

Legality Rules

4 A package_body shall be the completion of a previous package_declaration ♦. A library package_declaration ♦ shall not have a body unless it requires a body♦.

Static Semantics

5 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 ♦ 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. ♦ For a library package, the place is partially determined by the elaboration dependences (see Section 10).)

Dynamic Semantics

For the elaboration of a ♦ 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. ♦ 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 and 10.2, ''Program Execution''). 8

Examples

Example of a package body (see 7.1): 9

```
package body Rational_Numbers is                                          10

   procedure Same_Denominator (X,Y : in out Rational) is                  11
   begin
      -- reduces X and Y to the same denominator:
      ...
   end Same_Denominator;

   function Equal (X,Y : Rational) return Boolean is                      12
      U : Rational := X;
      V : Rational := Y;
   begin
      Same_Denominator (U,V);
      return U.Numerator = V.Numerator;
   end Equal;

   function Div (X,Y : Integer) return Rational is                        13
   begin
      if Y > 0 then
         return (Numerator => X,  Denominator => Y);
      else
         return (Numerator => -X, Denominator => -Y);
      end if;
   end Div;

   function Plus (X,Y : Rational) return Rational is ...  end Plus;       14
   function Minus (X,Y : Rational) return Rational is ...  end Minus;
   function Times (X,Y : Rational) return Rational is ...  end Times;
   function Div (X,Y : Rational) return Rational is ...  end Div;

end Rational_Numbers;                                                     15
```

7.3 Private Types

The declaration (in the visible part of a package) of a type as a private type ♦ 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). ♦ 1

Syntax

private_type_declaration ::= 2
  type defining_identifier **is private**;

♦ 3

Legality Rules

A private_type_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_ 4 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*. ♦

5 A type shall be completely defined before it is frozen (see 3.11.1 and 13.14). Thus ♦ the declaration of a variable of a partial view of a type ♦ is not allowed before the full declaration of the type. ♦

6 ♦

Abstract Syntax

7 A private type declaration has no type or specification.

$d_{pt} \in$ TypeDecl == **type** *id* **nil nil**

Static Semantics

14 A private_type_declaration declares a private type and its first subtype. ♦

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. 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 ♦ which components♦ are visible, what attributes and other predefined operations are allowed, and whether the first subtype is static. See 7.3.1.

16 ♦

Dynamic Semantics

17 The elaboration of a private_type_declaration 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. ♦

19 6 ♦

20 7 ♦

Examples

21 *Examples of private type declarations:*

22
```
type Key is private;
♦
```

7.3.1 Private Operations

1 For a type declared in the visible part of a package ♦, certain operations on the type do not become visible until later in the package — either in the private part or the body. Such *private operations* are available only inside the declarative region of the 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. ♦

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 within the immediate scope of the composite type 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. 3

NOTES
8 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♦, 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. 10

The consequences of this actual implementation are, however, valid everywhere. ♦ 11

9 Partial views provide assignment ♦, membership tests, ♦ qualification, and explicit conversion. 12

10 ♦ 13

Examples

Example of a type with private operations: 14

```
package Key_Manager is                                                    15
   type Key is private;
   Null_Key : constant Key; -- a deferred constant declaration (see 7.4)
   procedure Get_Key(K : in out Key);
   function Lt (X, Y : Key) return Boolean;
private
   type Key is  array (1..10 of Integer);
   Null_Key : constant Key := Key'(Others => 0);
end Key_Manager;

package body Key_Manager is                                               16
   Last_Key : Key := Null_Key;
   procedure Get_Key(K : in out Key) is
   begin
      Last_Key := Last_Key + 1;
      K := Last_Key;
   end Get_Key;

   function Lt (X, Y : Key) return Boolean is                             17
   begin
      return  X(1) < Y(1);
   end Lt;
end Key_Manager;
```

NOTES
11 *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 function Lt; they do not include other relational operators such as ">="♦. 18

♦ 19

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.) 20

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. ♦

Legality Rules

2 A *deferred constant declaration* is an object_declaration with the reserved word **constant** but no initialization expression. The constant declared by a deferred constant declaration is called a *deferred constant*. A deferred constant declaration requires a completion, which shall be a full constant declaration (called the *full declaration* of the deferred constant) ♦.

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 • The deferred and full constants shall have the same type;

6 • ♦

7 • ♦

8 ♦

9 The completion of a deferred constant declaration shall occur before the constant is frozen (see 7.4).

Dynamic Semantics

10 The elaboration of a deferred constant declaration ♦ has no effect.

NOTES

11 12 The full constant declaration for a deferred constant that is of a given private type ♦ is not allowed before the corresponding full_type_declaration. This is a consequence of the freezing rules for types (see 13.14).

Examples

12 *Examples of deferred constant declarations:*

13
```
Null_Key : constant Key;       -- see 7.3.1
```
14
```
CPU_Identifier : constant String(1..8);
♦
```

7.5 Limited Types -- Removed

7.6 Assignment and Finalization

1 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 ♦). Every object is finalized before being destroyed (for example, by leaving a subprogram_body containing an object_declaration♦). An assignment operation is used as part of assignment_statements, explicit initialization, parameter passing, and other operations.

2 Default definitions for these three fundamental operations are provided by the language.♦

Static Semantics

♦ 3

♦ The (default) implementations of Initialize♦ and Finalize have no effect. ♦ 4

Dynamic Semantics

♦ 11

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♦. ♦ 12

When a target object ♦ is assigned a value, either when created or in a subsequent assignment_statement, the *assignment operation* proceeds as follows: 13

- The value of the target becomes the assigned value. 14
- ♦ 15

♦

♦ 16

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. ♦ 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’’. ♦ 11

♦

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 ♦ but before leaving. Other constructs and entities are left immediately upon completion. 1

Dynamic Semantics

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. Completion due to reaching the end of execution, or due to the transfer of control of an exit_ or return_statement♦. Completion is *abnormal* otherwise — when control is transferred out of a construct due to abort or the raising of an exception. 2

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. Leaving an execution happens immediately after its completion, except in the case of a *master*: the execution of ♦ a block_statement or a subprogram_body♦. A master is finalized after it is complete, and before it is left. 3

FORMAL NOTES 4

The AVA definition of finalization of masters and objects implies that in all cases finalization has no effect. See (7) below. 5

6 ♦

7 For the *finalization* of a master, ♦ each object 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.

8 For the *finalization* of an object:

9 • If the object is of an elementary type, finalization has no effect;

10 • ♦

11 • ♦

12 • 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♦.

13 ♦

8. Visibility Rules

1 The rules defining the scope of declarations and the rules defining which identifiers, and character_ literals♦ 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. ♦

8.1 Declarative Region

Static Semantics

1 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;
- 5 ♦
- 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 ♦
- 11 ♦

12 The declarative region of a declaration is also called the *declarative region* of any view or entity declared by the declaration.

13 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.

14 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.

15 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.

8.2 Scope of Declarations

1 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.

Static Semantics

2 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♦). The immediate scope extends to the end of the declarative region, with the following exceptions:

3 • The immediate scope of a library_item includes only its semantic dependents.

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.

5 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. 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 • The visible part of a view of a callable entity is its profile.

7 • The visible part of a composite type♦ consists of the declarations of all components declared (explicitly or implicitly) within the type_declaration.

8 • ♦

9 • The visible part of a package♦ consists of declarations in the package's declaration other than those following the reserved word **private**, if any; see 7.1 ♦.

10 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.

11 The immediate scope of a declaration is also the immediate scope of the entity or view declared by the declaration. Similarly, the scope of a declaration is also the scope of the entity or view declared by the declaration.

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 ♦. 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

8.3 Visibility

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. 1

Static Semantics

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 ♦ is necessary to denote the declaration. 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. 2

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. 3

There are two kinds of direct visibility: *immediate visibility* and *use-visibility*. A declaration is immediately visible at a place if it is directly visible because the place is within its immediate scope. A declaration is use-visible if it is directly visible because of a use_clause (see 8.4). Both conditions can apply. 4

A declaration can be *hidden*, either from direct visibility, or from all visibility, within certain parts of its scope. Where *hidden from all visibility*, it is not visible at all (neither using a direct_name nor a selector_name). Where *hidden from direct visibility*, only direct visibility is lost; visibility using a selector_name is still possible. 5

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

The declarations of callable entities (including enumeration literals) are *overloadable*, meaning that overloading is allowed for them. ♦ 7

Two declarations are *homographs* if they have the same defining name, and, if both are overloadable, their profiles are type conformant. An inner declaration hides any outer homograph from direct visibility. 8

Two homographs are not ♦ allowed immediately within the same declarative region ♦. 9

A declaration is visible within its scope, except where hidden from all visibility, as follows: 10

- ♦ 11
- A declaration is hidden from all visibility until the end of the declaration, except: 12
  - For a record type ♦, the declaration is hidden from all visibility only until the reserved word **record**; 13
  - For a package_declaration♦ or subprogram_body, the declaration is hidden from all visibility only until the reserved word **is** of the declaration. ♦ 14

15 • 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 ♦ parameter_specification is hidden within the scope of a corresponding ♦ parameter_specification of a corresponding completion♦.

16 • The declaration of a library unit (including a library_unit_renaming_declaration) is hidden from all visibility except at places that are within its declarative region or within the scope of a with_clause that mentions it.

17 A declaration with a defining_identifier♦ is immediately visible (and hence directly visible) within its immediate scope except where hidden from direct visibility, as follows:

18 • 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;

19 • A declaration is also hidden from direct visibility where hidden from all visibility.

Name Resolution Rules

20 A direct_name shall resolve to denote a directly visible declaration whose defining name is the same as the direct_name. A selector_name shall resolve to denote a visible declaration whose defining name is the same as the selector_name. ♦

21 These rules on visibility and direct visibility do not apply in a context_clause or a parent_unit_name, ♦. For those contexts, see the rules in 10.1.6, ‘‘Environment-Level Visibility Rules’’.

Legality Rules

22 An explicit 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 explicit declaration. Similarly, the context_clause for a subunit 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 corresponding stub, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. ♦ ♦

NOTES

23 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.

24 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).

25 7 Not all contexts where an identifier or character_literal ♦ are allowed require visibility of a corresponding declaration. Contexts where visibility is not required are identified by using one of these ♦ syntactic categories directly in a syntax rule, rather than using direct_name or selector_name.

8.4 Use Clauses

1 A use_package_clause achieves direct visibility of declarations that appear in the visible part of a package♦.

Syntax

2 use_clause ::= use_package_clause | ♦

3 use_package_clause ::= **use** *package*_name {, *package*_name};

♦

Legality Rules

A *package*_name of a use_package_clause shall denote a package. 5

Static Semantics

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. ♦ 7

For each package denoted by a *package*_name of 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. ♦ 8

A declaration is *use-visible* if it is potentially use-visible, except in these naming-conflict cases: 9

- A potentially use-visible declaration is not use-visible if the place considered is within the immediate scope of a homograph of the declaration. 10
- Potentially use-visible declarations that have the same identifier are not use-visible unless each of them is an overloadable declaration. 11

Dynamic Semantics

The elaboration of a use_clause has no effect. 12

Examples

Example of a use clause in a context clause: 13

```
with Ada.Calendar; use Ada;
```
14

♦

♦

8.5 Renaming Declarations

A renaming_declaration declares another name for an entity, such as an object, ♦ package, or subprogram, ♦ but not an attribute. ♦ 1

Syntax

```
renaming_declaration ::=
    object_renaming_declaration
   | ♦
   | package_renaming_declaration
   | subprogram_renaming_declaration
   | ♦
```
2

Abstract Syntax

3

rename$_{pkg}$ ∈ RenamePackage == **rename-package** *id id*

$rename_p \in$ RenameSubprogram	== **rename-subprogram** *subprogram id*
$rename_o \in$ RenameObj	== **rename-obj** *id type id*
$rename \in$ Rename	== $rename_{pkg} \mid rename_p \mid rename_o$

Dynamic Semantics

4 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 (the *renamed view* and *renamed entity*). A name that denotes the renaming_declaration denotes (a new view of) the renamed entity.

NOTES

5 8 Renaming may be used to resolve name conflicts and to act as a shorthand. Renaming with a different identifier ♦ does not hide the old name; the new name and the old name need not be visible at the same places.

6 9 ♦

7 10 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

8
```
    subtype Mode is Ada.Text_IO.File_Mode;
```

8.5.1 Object Renaming Declarations

1 An object_renaming_declaration is used to rename an object.

Syntax

2 object_renaming_declaration ::= defining_identifier : subtype_mark **renames** *object*_name;

Name Resolution Rules

3 The type of the *object*_name shall resolve to the type determined by the subtype_mark.

Legality Rules

4 The renamed entity shall be an object.

5 ♦

Static Semantics

6 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♦

Examples

7 *Example of renaming an object:*

8
```
declare
   L : Person renames   Next_Person -- see 3.8
begin
   L.Age := L.Age + 1;
end;
```

8.5.2 Exception Renaming Declarations -- Removed

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

Static Semantics

A package_renaming_declaration declares a new view of the renamed package. 4

Examples

Example of renaming a package: 5

```
package TM renames Table_Manager;
```
6

8.5.4 Subprogram Renaming Declarations

♦ A subprogram_renaming_declaration ♦is called a *renaming-as-declaration*, and is used to rename a subprogram (possibly an enumeration literal)♦. ♦ 1

Syntax

subprogram_renaming_declaration ::= subprogram_specification **renames** *callable_entity_*name; 2

Name Resolution Rules

The expected profile for the *callable_entity_*name is the profile given in the subprogram_specification. 3

Legality Rules

The profile of a renaming-as-declaration shall be mode-conformant with that of the renamed callable entity. 4

♦ 5

A name that denotes a formal parameter of the subprogram_specification is not allowed within the *callable_entity_*name. 6

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 ♦ from the profile given in the subprogram_renaming_declaration. The new view is a function or procedure♦. ♦ 7

♦

NOTES

11 A procedure can only be renamed as a procedure. A function♦ can be renamed only with an identifier ♦. Enumeration literals can be renamed as functions♦ Attributes and operators cannot be renamed. 9

12 ♦ 10

Examples

13 *Examples of subprogram renaming declarations:*

14
```
♦
function No_Free_Space (Foo : Integer) return Boolean renames Free_List_Empty;
```

15 ♦

16 ♦

17 ♦

8.5.5 Generic Renaming Declarations -- Removed

8.6 The Context of Overload Resolution

1 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 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 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.

Name Resolution Rules

4 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.

7 • A statement.

8 • ♦

9 • The expression of a case_statement.

10 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

12 • for each usage name, which declaration it denotes (and, therefore, which view and which entity it denotes); and ♦

- 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

A *possible interpretation* is one that obeys the syntax rules and the visibility rules. 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

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

Each occurrence of a usage name *denotes* the declaration determined by its interpretation. It also denotes the view declared by its denoted declaration.♦ ♦ 16

A usage name that denotes a view also denotes the entity of that view. 19

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. The type resolution rules provide support for ♦ universal numeric literals♦: 20

- 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
- 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
  - ♦ 23
  - to a universal type that covers *T*; ♦ 24
  - ♦ 25

In certain contexts, such as in a subprogram_renaming_declaration, the Name Resolution Rules define an *expected profile* for a given name; in such cases, the name shall resolve to the name of a callable entity whose profile is type conformant with the expected profile. 26

Legality Rules

When the expected type for a construct is required to be a *single* type in a given class, the type expected for 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; the type of the construct is then this single expected type. 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. ♦ 27

A complete context shall have at least one acceptable interpretation; if there is exactly one, then that one is chosen. 28

There is a *preference* for the primitive operators (and ranges) of the root numeric type *root_integer* ♦. 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* ♦, and the other is not, the interpretation using the primitive operator (or range) of the root numeric type is *preferred*. 29

30 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. Otherwise, the complete context is *ambiguous*.

31 A complete context ♦ shall not be ambiguous.

32 ♦

NOTES

33 13 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.

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).

9. Tasks and Synchronization -- Removed

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 ♦ partition which may execute in a separate address space♦. 1

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. 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

10.1 Separate Compilation

A *program unit* is either a package♦ 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

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 or the body ♦. The representation for a compilation is implementation-defined. 2

A library unit is a separately compiled program unit, and is always a package or a subprogram♦. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a *subsystem*. 3

Implementation Permissions

An implementation may impose implementation-defined restrictions on compilations that contain multiple compilation_units. 4

10.1.1 Compilation Units - Library Units

A library_item is a compilation unit that is the declaration or body♦ of a library unit. Each library unit (except Standard) has a *parent unit*, which is a library package♦. A library unit is a *child* of its parent unit. The *root* library units are the children of the predefined library package Standard. 1

Syntax

```
compilation ::= {compilation_unit}                                2

compilation_unit ::=                                              3
   context_clause library_item
  | ♦

library_item ::= ♦ library_unit_declaration                      4
  | library_unit_body
  | ♦

library_unit_declaration ::=                                      5
    subprogram_declaration | package_declaration
   | ♦
```

6 ♦

7 library_unit_body ::= subprogram_body | package_body

8 parent_unit_name ::= name

9 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 ♦, as well as to its declaration and body, as in ‘‘library procedure’’ or ‘‘library package_body’’♦. 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 ♦.

10 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. 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 ♦ is a *root* library unit ♦. The declaration and body of Standard itself have no parent declaration. The *parent unit* of a library_item or library unit is the library unit declared by its parent declaration.

11 The children of a library unit occur immediately within the declarative region of the declaration of the 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.) The *descendant* relation is the inverse of the ancestor relation.

12 A library_unit_declaration ♦ is ♦ *public*. ♦ The *public descendants* of a library unit are the library unit itself, and the public descendants of its public children. ♦

Legality Rules

13 The parent unit of a library_item shall be a library package ♦.

14 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. ♦

15 A parent_unit_name (which can be used within a defining_program_unit_name of a library_item ♦) and each of its prefixes, shall not denote a renaming_declaration. ♦

16 ♦ ♦

17 ♦

Abstract Syntax

18

with	== *id**
use	== *id**
compilation ∈ Compilation	== **compilation** *comp-unit**
comp-unit ∈ CompUnit	== **comp-unit** *library-unit context*
context ∈ Context	== **context** [*with*] [*use*]

Static Semantics

19 ♦ There are two kinds of dependences among compilation units:

- 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. 20
- The *elaboration dependences* (see 10.2) determine the order of elaboration of library_items. 21

♦

A library_item depends semantically upon its parent declaration. ♦ A library_unit_body depends semantically upon the corresponding library_unit_declaration, if any. A compilation unit depends semantically upon each library_item mentioned in a with_clause of the compilation unit. The semantic dependence relationship is transitive. ♦ 22

NOTES
1 A simple program may consist of a single compilation unit. A compilation need not have any compilation units ♦. 23

2 ♦ 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. 24

♦

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

Syntax

context_clause ::= {context_item} 2

context_item ::= with_clause | use_clause 3

with_clause ::= **with** *library_unit_*name {, *library_unit_*name}; 4

Name Resolution Rules

The *scope* of a with_clause that appears on a library_unit_declaration ♦ consists of the entire declarative region of the declaration, which includes all children ♦. The scope of a with_clause that appears on a body consists of the body♦. 5

A library_item is *mentioned* in a with_clause if it is denoted by a *library_unit_*name or a prefix in the with_clause. 6

Outside its own declarative region, the declaration ♦ of a library unit can be visible only within the scope of a with_clause that mentions it. The visibility of the declaration ♦ of a library unit otherwise follows from its placement in the environment. 7

♦

NOTES
3 A library_item mentioned in a 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, and a library subprogram can be called ♦. 9

10.1.3 Subunits of Compilation Units -- Removed

10.1.4 The Compilation Process

1 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’’).

2 The declarative_items of the environment are library_items appearing in an order such that there are no forward semantic dependences. ♦ 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 The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined.

Name Resolution Rules

4 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 for a subprogram ♦ with the same defining_program_unit_name already exists in the environment (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.

Legality Rules

5 When a compilation unit is compiled, all compilation units upon which it depends semantically shall already exist in the environment; 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.

Implementation Permissions

6 The implementation must require that a compilation unit be legal before inserting it into the environment.

7 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 with the same defining_program_unit_name. ♦ 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. ♦

NOTES

8 4 The rules of the language are enforced across compilation and compilation unit boundaries, just as they are enforced within a single compilation unit.

9 5 An implementation may support a concept of a *library*, which contains library_items and their subunits. 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.

10 6 ♦

10.1.5 Pragmas and Program Units -- Removed

10.1.6 Environment-Level Visibility Rules

1 The normal visibility rules do not apply within a parent_unit_name or a context_clause, ♦. The special visibility rules for those contexts are given here. ♦

Static Semantics

2 Within the parent_unit_name at the beginning of a library_item, and within a with_clause, the only declarations that are visible are those that are library_items of the environment, and the only declarations that are directly visible are those that are root library_items of the environment.

3 ♦ Within a use_clause ♦ 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.

4 ♦

10.2 Program Execution

1 An Ada *program* consists of a single partition.

Post-Compilation Rules

2 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. 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 ♦ or by some ♦ implementation-defined means): ♦ ♦

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 • ♦

7 The user must designate (in an implementation-defined manner) one subprogram as the *main* subprogram for the partition. A main subprogram, if specified, shall be a subprogram.

8 ♦A 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. ♦

9 *Program execution* consists of the execution of the partition that defines the program.

10 The order of elaboration of library units is determined primarily by the *elaboration dependences*. There is an elaboration dependence of a given library_item upon another if the given library_item ♦ depends semantically on the other library_item. ♦

10 It is required that when a library_item is a library_unit_body, the item is elaborated *immediately* after the library_unit which it completes. This avoids certain erroneous programs based on order of elaboration. It has some other effects.

11 • It rules out mutual recursion between routines defined in different packages.

12 • If package specification B depends on package A, then the package body of A (if there is one) is elaborated before package specification B.

13 • A package specification is always elaborated before its body.

14 ♦

18 There shall be a total order of the library_items that obeys the above rules. The order is otherwise implementation defined. ♦

19 The full expanded names of the library units ♦ included in a given partition shall be distinct.

20 ♦

♦ 21

The mechanisms for building and running a partition are implementation defined. These might be combined into one operation, as, for example, in dynamic linking, or ‘‘load-and-go’’ systems. 24

Dynamic Semantics

The execution of a program consists of the execution of its partition. Further details are implementation defined. The execution of a partition starts with the execution of its environment task and ends when the environment task terminates♦ ♦ 25

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

Implementation Permissions

♦ 28

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

♦ 30

10.2.1 Elaboration Control -- Removed

11. Exceptions

This section defines the facilities for dealing with errors or other exceptional situations that arise during program execution. An *exception* represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an *exception occurrence*. To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called *handling* the exception. 1

There is no facility in AVA for programmer-defined exceptions. 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. If no handler exists, the exception is propagated out to the main program. AVA places extremely onerous restrictions on the Ada exception handling mechanism. What remains is intended to allow routines to handle exceptions, reinitialize themselves, and continue. We have attempted to make it impossible for AVA programs to use the exception mechanism to distinguish between different implementation choices in those places where operations may be performed in an arbitrary order. 2

11.1 Exception Declarations

♦

Abstract Syntax

1

exc ∈ Exception == **exception** *id*

Static Semantics

♦ 2

The *predefined* exceptions are the ones declared in the declaration of package Standard: Constraint_Error, Program_Error, and Storage_Error♦; one of them is raised when a language-defined check fails. ♦ 4

Dynamic Semantics

♦ 5

The execution of any construct raises Storage_Error if there is insufficient storage for that execution. The amount of storage needed for the execution of constructs is unspecified. Since storage error can occur at so many program locations in an implementation dependent fashion we are forced to ignore it in the formal semantics. All program proof will therefore be predicated on the assumption that storage error does not occur. 6

♦

11.2 Exception Handlers

The response to one or more exceptions is specified by an exception_handler. 1

Syntax

2 handled_sequence_of_statements ::=
  sequence_of_statements
 [**exception**
  exception_handler
  ♦]

3 exception_handler ::=
 when ♦ exception_choice ♦ =>
  sequence_of_statements

♦

5 exception_choice ::= ♦ **others**

Legality Rules

6 An exception_choice can only be **others**. It covers all exceptions. ♦

Abstract Syntax

7 *handler* ∈ Handler == $stmt^*$

♦

Dynamic Semantics

10 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.

Examples

11 *Example of an exception handler:*

12
```
begin
   x := y/z;        -- sequence of statements
exception
   when others =>
      Put(Standard_Output, "Fatal Error");
      raise Program_Error;
end;
```

♦

11.3 Raise Statements

1 A raise_statement raises an exception.

Syntax

2 raise_statement ::= **raise** Program_Error;

Legality Rules

3 ♦The only exception that may appear in a raise_statement is Program_Error.

Abstract Syntax

4 *raise* ∈ Raise == **raise**

Dynamic Semantics

To *raise an exception* is to raise a new occurrence of that exception, as explained in 11.4. For the execution of a raise_statement ♦, the ♦ exception Program_Error is raised. ♦ ♦ 5

Examples

Example of raise statements: 6

```
♦                                                                    7
raise Program_Error;                                                 8
```

♦

11.4 Exception Handling

When an exception occurrence is raised, normal program execution is abandoned and control is transferred to an ♦ exception_handler, if any. To *handle* an exception occurrence is to respond to the exceptional event. 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

Dynamic Semantics

♦ If the execution of construct *a* is defined by this Reference Manual 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*. The *innermost dynamically enclosing* execution of a given execution is the dynamically enclosing execution that started most recently. 2

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: 3

- ♦ 4
- If the construct is the sequence_of_statements of a handled_sequence_of_statements that has a handler ♦, the occurrence is handled by that handler; 5
- Otherwise, the occurrence is *propagated* to the innermost dynamically enclosing execution, which means that the occurrence is raised again in that context. 6

♦ When an occurrence is *handled* by a given handler, ♦ the sequence_of_statements of the handler is executed; this execution replaces the abandoned portion of the execution of the sequence_of_statements. 7

NOTES

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. 8

11.4.1 The Package Exceptions -- Removed

11.4.2 Example of Exception Handling

Examples

Exception handling may be used to separate the detection of an error from the response to that error: 1

2
```
function Factorial (N : Positive) return Integer is
begin
  if N = 1 then
    return 1 ;
  else
    return N * Factorial(N-1);
  end if;
exception
  when others => return Integer'Last;
end Factorial;
```

3 If the multiplication raises Constraint_Error, then Integer'Last is returned by the handler. This value will cause further Constraint_Error exceptions to be raised by the evaluation of the expression in each of the remaining invocations of the function, so that for large values of N the function will ultimately return the value Integer'Last.

4 It will be difficult to predict the behavior of programs that depend on particular values of potentially affected global or local variables within the scope of the frame when control is transfered to an **others** handler. For safe programming, any such variables that the program depends on should be reinitialized in the handler.

5
```
package P is
  procedure R;
  procedure Q;
end P;
```

6
```
package body P is
  procedure Q is
  begin
    R;
    ...          --  error situation (2)
  exception
    when others =>  --  handler E2
    ...
  end Q;
  procedure R is
    begin
      ...          --  error situation (3)
    end R;
```

7
```
begin
  ...            --  error situation (1)
  Q;
  ...
exception
   when others =>     --  handler E1
  ...
end P;
```

The following situations can arise:

8 1. If the exception Program_Error is raised in the sequence of statements of the outer package P, the handler E1 provided within P is used to complete the execution of P.

9 2. If the exception Program_Error is raised in the sequence of statements of Q, the handler E2 provided within Q is used to complete the execution of Q. Control will be returned to the point of call of Q upon completion of the handler.

10 3. If the exception Program_Error is raised in the body of R, called by Q, the execution of R is abandoned and the same exception is raised in the body of Q. The handler E2 is then used to complete the execution of Q, as in situation (2).

Note that in the third situation, the exception raised in R results in (indirectly) transferring control to a handler that is part of Q and hence not enclosed by R. Note also that if a handler were provided within R for the exception choice **others**, situation (3) would cause execution of this handler, rather than direct termination of R.

11.5 Suppressing Checks -- Removed

11.6 Exceptions and Optimization -- Removed

12. Generic Units -- Removed

13. Representation Issues

This section describes features for querying and controlling aspects of representation and for interfacing to hardware. 1

♦

13.1 Representation Items -- Removed

13.2 Pragma Pack -- Removed

13.3 Representation Attributes -- Removed

13.4 Enumeration Representation Clauses -- Removed

13.5 Record Layout -- Removed

13.6 Change of Representation -- Removed

13.7 The Package System

For each implementation there is a predefined library package called SYSTEM which includes the definitions of certain configuration-dependent characteristics. 1

Static Semantics

The following language-defined library package exists: 2

3

```
package SYSTEM is                                                        4
   ♦
   type Name is implementation_defined_enumeration_type;
   AVA_System_Name : constant Name := implementation_defined;            5
   ♦                                                                     6
   --  System-Dependent Named Numbers:                                   7
   AVA_Min_Int   : constant := implementation_defined;                   8
   AVA_Max_Int   : constant := implementation_defined;
   ♦
end SYSTEM;                                                              9
```

Values of the enumeration type Name are the names of alternative machine configurations handled by the implementation; one of these is the constant AVA_System_Name. ♦ 10

NOTES

1 It is a consequence of the visibility rules that a declaration given in the package STANDARD is not visible in a compilation unit unless this package is mentioned by a with clause that applies (directly or indirectly) to the compilation unit. 11

13.8 Machine Code Insertions -- Removed

13.9 Unchecked Type Conversions -- Removed

13.10 Unchecked Access Value Creation -- Removed

13.11 Storage Management -- Removed

13.12 Pragma Restrictions -- Removed

13.13 Streams -- Removed

13.14 Freezing Rules

1 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.

2 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).

3 The end of a declarative_part ♦ or a declaration of a library package ♦ causes *freezing* of each entity declared within it ♦. A ♦ body causes freezing of each entity declared before it within the same declarative_part.

4 A construct that (explicitly or implicitly) references an entity can cause the *freezing* of the entity, as defined by subsequent paragraphs. At the place where a construct causes freezing, each name, expression, or range within the construct causes freezing:

5 ♦

6 The occurrence of an object_declaration that has no corresponding completion causes freezing.

7 ♦

8 A static expression causes freezing where it occurs. A nonstatic expression causes freezing where it occurs♦.

9 The following rules define which entities are frozen at the place where a construct causes freezing:

10 At the place where an expression causes freezing, the type of the expression is frozen♦. ♦

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; At the place where an object name causes freezing, the nominal subtype associated with the name is frozen. 11

At the place where a range causes freezing, the type of the range is frozen. 12

♦ 13

At the place where a callable entity is frozen, each subtype of its profile is frozen. ♦ 14

At the place where a subtype is frozen, its type is frozen. 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. ♦ 15

Legality Rules

♦ 16

A type shall be completely defined before it is frozen (see 3.11.1). ♦ 17

The completion of a deferred constant declaration shall occur before the constant is frozen (see 7.4). 18

♦ 19

♦

The Standard Libraries

A. Predefined Language Environment

This Annex contains the specifications of library units that shall be provided by every implementation. There are two root library units: Ada ♦ and System; } other library units are children of these: 1

Standard — A.1
    Ada — A.2
        AVA_IO — A.10.1
    System — 13.7

♦

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

A.1 The Package Standard

This clause outlines the specification of the package Standard containing all predefined identifiers in the language. 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 are used for pseudo-names of anonymous types (such as *root_integer*) and for undefined information (such as *implementation-defined*). 2

Static Semantics

The library package Standard has the following declaration: 3

```
package Standard is                                                    4
♦
   type Boolean is (False, True);                                      5
   -- The predefined relational operators for this type are as follows:   6
   -- function "="   (Left, Right : Boolean) return Boolean;           7
   -- function "/="  (Left, Right : Boolean) return Boolean;
   -- function "<"   (Left, Right : Boolean) return Boolean;
   -- function "<="  (Left, Right : Boolean) return Boolean;
   -- function ">"   (Left, Right : Boolean) return Boolean;
   -- function ">="  (Left, Right : Boolean) return Boolean;
   -- The predefined logical operators and the predefined logical      8
   -- negation operator are as follows:
   -- function "and" (Left, Right : Boolean) return Boolean;           9
   -- function "or"  (Left, Right : Boolean) return Boolean;
   -- function "xor" (Left, Right : Boolean) return Boolean;
   -- function "not" (Right : Boolean) return Boolean;                 10
   -- The integer type root_integer is predefined.                     11
   -- The corresponding universal type is universal_integer.
   -- The integer type root_integer is predefined.
   -- Note that AVA doesn't permit the syntax below for the definition of Integer.
   type Integer is range implementation-defined;                       12
   subtype Natural  is Integer range 0 .. Integer'Last;                13
   subtype Positive is Integer range 1 .. Integer'Last;
   -- The predefined operators for type Integer are as follows:        14
```

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 ♦

21
```
    -- Note that AVA doesn't provide Float, but we insert the declaration below
    -- in order to detect conflicts in attempted use.  Also, the use of limited
    -- is outside the scope of the AVA language, but interpreted in the Ada sense
    -- and the AVA declaration restrictions ensures that we cannot create a
    -- variable of the type.
```
22
```
    type Float is  limited private;
```

35

36
```
       -- The declaration of type Character is based on the standard ISO 8859-1 character set.
```
37
```
       -- 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#)

-- The predefined operators for the type Character are the  same  as  for      38
-- any enumeration type.
```

```
♦
type Wide_Character is limited private;

package ASCII is ... end ASCII;  --Obsolescent; see I.5

-- Predefined string types:
type String is array(Positive range <>) of Character;                          39
♦
-- The predefined operators for this type are as follows:                      40
--     function "="  (Left, Right: String) return Boolean;                     41
--     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;         42
--     function "&" (Left: Character; Right: String)    return String;
--     function "&" (Left: String;    Right: Character) return String;
--     function "&" (Left: Character; Right: Character) return String;

♦                                                                               43
type Wide_String is limited private;                                           44
                                                                               45
-- The predefined exceptions:

♦                                                                               45
Program_Error   : exception;
♦
end Standard;                                                                  46
```

Standard has no private part. 47

In ♦the type 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 Reference Manual 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 Reference Manual refers to the hyphen character. 48

Dynamic Semantics

49 Elaboration of the body of Standard has no effect. ♦

♦

Implementation Advice

52 ♦

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 ♦ The name of a library unit cannot be a homograph of a name (such as Integer) that is already declared in Standard.

56 4 ♦

A.2 The Package Ada

Static Semantics

1 The following language-defined library package exists:

2
```
package Ada is
    ♦
end Ada;
```

3 Ada serves as the parent of most of the other language-defined library units; its declaration is empty ♦.

Legality Rules

4 In the standard mode, it is illegal to compile a child of package Ada. ♦

A.3 Character Handling -- Removed

A.4 String Handling -- Removed

A.5 The Numerics Packages -- Removed

A.6 Input-Output

1 Input-output is provided through language-defined packages, each of which is a child of the root package Ada. Operations for text input-output are supplied in the package AVA-IO.[11]

[11]Note that this package should be trivially implementable using Ada’s language-defined Text_IO. The reason we define a completely new package is in order to avoid turning ambiguous Ada programs into unambiguous AVA programs. Text_IO makes extensive use of default parameters, which we have excluded from our subset.

A.7 External Files and File Objects

Static Semantics

1 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 in a system-dependent fashion

2 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. File objects are essentially indices into tables maintained by the AVA_IO package. As such, they are always passed to predefined routines as constant (**in**) parameters. Any actual changes occur in these internal tables.

3 ♦

5 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*. This association is accomplished in an implementation-dependent manner. The objects of file type that are passed into the main program and declared in AVA_IO are already open. Once closed they cannot be reopened.

6 The language does not define what happens to external files after the completion of the main program♦ (in particular, if corresponding files have not been closed). ♦

7 An open file has a *current mode*, which is a value of ♦ the following enumeration type:

```
8    type File_Mode is (In_File, ♦ Out_File);  -- for AVA_IO
```

9 These values correspond respectively to the cases where only reading♦ or only writing are to be performed. ♦

10 ♦

12 The mode of a file cannot be changed.

13 ♦

14 ♦ The only exception that can be raised by a call of an input-output subprogram is Program_Error;[12] the situations in which it can be raised are described, either following the description of the subprogram or in Appendix L in the case of error situations that are implementation-dependent.

NOTES
15 5 ♦

[12]To help the reader relate these exceptions to those defined for Text_IO in Ada, we use the notation $\text{Program\_Error}_{\text{original}}$ to indicate what original exception was replaced by Program_Error. E.g., $\text{Program\_Error}_{\text{status}}$, $\text{Program\_Error}_{\text{mode}}$, ...

A.8 Sequential and Direct Files

Static Semantics

1 One kind of access to external files is defined in this subclause: *sequential access* ♦. ♦ A file object to be used for sequential access is called a *sequential file*♦. ♦.

2 For sequential access, the external 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. ♦

5 NOTES

6 6 A capability for appending to a file is a system-dependent property. In particular it depends on the manner in which external files are associated with parameters to the main program. See [AI-00278] for discussion in the context of full Ada.

A.8.1 The Generic Package Sequential_IO

Relevant portions moved to subclause A.8.2.

A.8.2 File Management

Static Semantics

1 The procedures and functions described in this subclause provide for the control of external files. ♦ The only allowed file modes for text files are the modes In_File and Out_File. ♦

9
```
procedure Close(File : in out File_Type);
```
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, outputs a file terminator.♦

The exception $\text{Program\_Error}_{\text{status}}$ is propagated if the given file is not open.

10 ♦

18
```
function Mode(File : in File_Type) return File_Mode;
```
Returns the current mode of the given file, either In_File or Out_File.

The exception $\text{Program\_Error}_{\text{status}}$ is propagated if the file is not open.

19 ♦

27
```
function Is_Open(File : in File_Type) return Boolean;
```
Returns True if the file is open (that is, if it is associated with an external file), otherwise returns False.

30

31
```
function End_Of_File(File : in File_Type) return Boolean;
```

32 Operates on a file of mode In_File. Returns True if no more elements can be read from the given file; otherwise returns False.

The exception $\text{Program\_Error}_{\text{mode}}$ is propagated if the mode is not In_File. The exception $\text{Program\_Error}_{\text{status}}$ is propagated if the file is not open.

♦

A.8.3 Sequential Input-Output Operations

Relevant portions moved to subclause A.8.2.

A.8.4 The Generic Package Direct_IO -- Removed

A.8.5 Direct Input-Output Operations -- Removed

A.9 The Generic Package Storage_IO -- Removed

A.10 Text Input-Output

Static Semantics

This clause further describes the package AVA_IO facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of characters ♦ 1

The facilities for file management given above, in subclause A.8.2 ♦, are available for text input-output. ♦ There are also procedures Get and Put that input values of types Character and String 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. ♦ 2

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. ♦ 5

♦ 6

From a logical point of view, a text file is a sequence of ♦ characters♦. One character constant is provided to mark the end of a line, EOL. The terminator is generated during output; either by calls of procedures provided expressly for that purpose♦ or by passing the value of this constant as a character to be output.[13] 7

♦ 8

When a file is initially open with mode Out_File, its size is unbounded. Storage_Error is propagated if external file size limits are encountered. 12

♦

A.10.1 The Package AVA_IO

Static Semantics

The library package AVA_IO has the following declaration: 1

```
    package AVA_IO is
```
2

[13]This means that an entire file can be copied by Get-ting and Put-ting characters, without any knowledge of the underlying file structure.

```
   type File_Type is private;

   type File_Mode is (In_File, Out_File);

   -- File Management

   procedure Close  (File : in out File_Type);
   function  Mode   (File : in File_Type) return File_Mode;
   function  Is_Open(File : in File_Type) return Boolean;
   function  End_Of_File(File : in File_Type) return Boolean;

   -- Standard input and output files

   Standard_Input  : constant File_Type;  -- Why not constant functions?
   Standard_Output : constant File_Type;

   -- Line Control

   EOL : constant CHARACTER;

   -- Character Input-Output

   procedure Get(File : in  File_Type; Item : in out Character);
   procedure Put(File : in  File_Type; Item : in Character);

   -- String Input-Output

   procedure Get(File : in  File_Type; Item : in out String);
   procedure Put(File : in  File_Type; Item : in String);

   procedure Get_Line(File : in  File_Type; Item : in out String; Last : in out Natural);
   procedure Put_Line(File : in  File_Type; Item : in String);

   -- Exceptions:
   -- These exceptions are not defined in AVA.  They are raised as Program_Error.
   -- But they are documented here in order that we can maintain the sense of
   -- the various reasons for exceptions raised by predefined I/O operations.

   -- Status_Error : exception;
   -- Mode_Error   : exception;
   -- Name_Error   : exception;
   -- Use_Error    : exception;
   -- Device_Error : exception;
   -- End_Error    : exception;
   -- Data_Error   : exception;
   -- Layout_Error : exception;

private
   -- implementation-dependent
end AVA_IO;
```

A.10.2 Text File Management

See subclause A.8.2.

A.10.3 Default Input, Output, and Error Files

The following constants provide one means to access the file pointers to the standard input and output files.

7
```
      Standard_Input : constant File_Type;
```

Value is a pointer to the standard input file.

9
```
      Standard_Output : constant File_Type;
```

Value is a pointer to the standard output file.

A.10.4 Specification of Line and Page Lengths -- Removed

A.10.5 Operations on Columns, Lines, and Pages

The end of line constant described in this subclause and the procedures Put_Line and Get_Line described in subclause A.10.6 provide the only explicit control of line structure in files.

```
EOL: constant Character := implementation_dependent    1
```

A.10.6 Get and Put Procedures

Static Semantics

1 The procedures Get and Put for items of the types Character and String♦ are described in subsequent subclauses. Features of these procedures that are common to ♦ these types are described in this subclause. The Get and Put procedures for items of type Character and String deal with sequences of individual character values♦.

2 All procedures Get and Put have forms with a file parameter, written first. ♦ Each procedure Get operates on a file of mode In_File. Each procedure Put operates on a file of mode Out_File ♦.

3 ♦

9 The exception Program_Error$_{status}$ 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 Program_Error$_{mode}$ 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 ♦.

10 The exception Program_Error$_{end}$ is propagated by a Get procedure if an attempt is made to skip a file terminator. ♦

A.10.7 Input-Output of Characters and Strings

Static Semantics

1 For an item of type Character the following procedures are provided:

```
procedure Get(File : in File_Type; Item : out Character);[14]    2
♦
```

3 ♦ Reads the next character from the specified input file and returns the value of this character in the in out parameter Item.

The exception Program_Error$_{end}$ is propagated if an attempt is made to read past the end of a file.

4

```
procedure Put(File : in File_Type; Item : in Character);    5
♦
```

6 ♦ Outputs the given character to the file.

[14]Because of the constraints that AVA places on parameter modes, excluding mode **out**, this and subsequent Ada **out** parameters are instead of **in out** mode.

7 ♦

10 For an item of type String the following procedures are provided:

11
```
procedure Get(File : in File_Type; Item : in out String);
♦
```

12 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).

13

14
```
procedure Put(File : in File_Type; Item : in String);
♦
```

15 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).

16

17
```
procedure Get_Line(File : in File_Type; Item : in out String; Last : in out Natural);
♦
```

18 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 EOL is met ♦ or if the end of file is encountered. If an EOL ended the Get_Line, it is skipped. The values of characters not assigned are left unchanged.

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 0). If no characters are read, returns in Last an index value that is one less than 0. The exception Program_Error$_{end}$ is propagated if an attempt is made to read past the end of a file.

19

20
```
procedure Put_Line(File : in File_Type; Item : in String);
♦
```

21 Calls the procedure Put for the given string, and then outputs an EOL.

♦

NOTES

21 7 ♦

22 8 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).

23 9 ♦

24 10 End of lines encountered by a Get will be skipped over, whicle Put may insert a number of them in the course of outputting a string.

A.10.8 Input-Output for Integer Types -- Removed

A.10.9 Input-Output for Real Types -- Removed

A.10.10 Input-Output for Enumeration Types -- Removed

A.11 Wide Text Input-Output -- Removed

A.12 Stream Input-Output -- Removed

A.13 Exceptions in Input-Output

♦The following exceptions are included for expository reasons. They cannot be raised by input-output operations since they have all been renamed to PROGRAM_ERROR. PROGRAM_ERROR is raised in AVA_IO at the points where the io exceptions were raised in TEXT_IO. Only outline descriptions are given of the conditions under which exceptions are raised; for full details see Appendix L. 1

Static Semantics

♦ 2

```
    package Ada.IO_Exceptions is                                    3

       ♦
       Status_Error : exception;
       Mode_Error   : exception;
       Name_Error   : exception;
       Use_Error    : exception;
       Device_Error : exception;
       End_Error    : exception;
       Data_Error   : exception;
       Layout_Error : exception;

    end Ada.IO_Exceptions;
```

If more than one error condition exists, the corresponding exception that appears earliest in the following list is the one that is propagated. 4

The exception Program_Error$_{status}$ 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. 5

The exception Program_Error$_{mode}$ is propagated by an attempt to read from, or test for, the end of a file whose current mode is Out_File ♦, and also by an attempt to write to a file whose current mode is In_File. ♦ 6

♦ 7

The exception Program_Error$_{use}$ is propagated if an operation is attempted that is not possible for reasons that depend on characteristics of the external file. ♦ 8

The exception Program_Error$_{device}$ is propagated if an input-output operation cannot be completed because of a malfunction of the underlying system. 9

10 The exception Program_Error$_{end}$ is propagated by an attempt to skip (read past) the end of a file.

11 ♦

Documentation Requirements

12 The implementation shall document the conditions under which Program_Error$_{name}$, Program_Error$_{use}$ and Program_Error$_{device}$ are propagated.

A.14 File Sharing -- Removed

A.15 The Package Command_Line -- Removed

B. Interface to Other Languages -- Removed

C. Systems Programming -- Removed

D. Real-Time Systems -- Removed

E. Distributed Systems -- Removed

F. Information Systems -- Removed

G. Numerics -- Removed

H. Safety and Security

Removed.

I. Obsolescent Features

This Annex contains descriptions of features of the language whose functionality is largely redundant with other features defined by this Reference Manual. Use of these features is not recommended in newly written programs. 1

♦

I.1 Renamings of Ada 83 Library Units -- Removed

I.2 Allowed Replacements of Characters -- Removed

I.3 Reduced Accuracy Subtypes -- Removed

I.4 The Constrained Attribute -- Removed

I.5 ASCII

Static Semantics

The following declaration exists in the declaration of package Standard: 1

```
package ASCII is                                                                   2
    -- Control characters:                                                         3
                                                                                   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;
    -- Other characters:                                                           5
    Exclam    : constant Character:= '!';  Quotation : constant Character:= '"';   6
    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:= '~';
    -- Lower case letters:                                                         7
```

8
```
            LC_A: constant Character:= 'a';
            ...
            LC_Z: constant Character:= 'z';
```
9
```
        end ASCII;
    ;
```

I.6 Numeric_Error -- Removed

I.7 At Clauses -- Removed

I.7.1 Interrupt Entries -- Removed

I.8 Mod Clauses -- Removed

I.9 The Storage_Size Attribute -- Removed

J. Language-Defined Attributes

This annex summarizes the definitions given elsewhere of the language-defined attributes. 1

S'Base — For every scalar subtype S: 2

S'Base denotes an unconstrained subtype of the type of S. See 3.5(15). 3

A'First(N) — For a prefix A that is of an array type ♦, or denotes a constrained array subtype: 4

A'First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type. See 3.6.2(3). 5

A'First — For a prefix A that is of an array type ♦, or denotes a constrained array subtype: 6

A'First denotes the lower bound of the first index range; its type is the corresponding index type. See 3.6.2(2). 7

S'First — For every scalar subtype S: 8

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(12). 9

S'Image — For every scalar subtype S: 10

S'Image denotes a function with the following specification: 11

12
```
function S'Image(Arg : S'Base)
  return String
```

The function returns an image of the value of *Arg* as a String. See 3.5(34). 13

A'Last(N) — For a prefix A that is of an array type ♦, or denotes a constrained array subtype: 14

A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type. See 3.6.2(5). 15

A'Last — For a prefix A that is of an array type ♦, or denotes a constrained array subtype: 16

A'Last denotes the upper bound of the first index range; its type is the corresponding index type. See 3.6.2(4). 17

S'Last — For every scalar subtype S: 18

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(13). 19

A'Length(N) — For a prefix A that is of an array type ♦, or denotes a constrained array subtype: 20

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(9). 21

A'Length — For a prefix A that is of an array type ♦, or denotes a constrained array subtype: 22

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(8). 23

S'Pos — For every discrete subtype S: 24

S'Pos denotes a function with the following specification: 25

26
```
function S'Pos(Arg : S'Base)
  return universal_integer
```

This function returns the position number of the value of *Arg*, as a value of type *universal_integer*. See 3.5.5(1). 27

S'Pred — For every scalar subtype S: 28

S'Pred denotes a function with the following specification: 29

30
```
function S'Pred(Arg : S'Base)
  return S'Base
```

31 For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; 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*. ♦ Constraint_Error is raised if there is no such machine number. See 3.5(24).

32 A'Range(N)	For a prefix A that is of an array type ♦, or denotes a constrained array subtype:
33	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(7).
34 A'Range	For a prefix A that is of an array type ♦, or denotes a constrained array subtype:
35	A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once. See 3.6.2(6).
36 S'Succ	For every scalar subtype S:
37	S'Succ denotes a function with the following specification:

38
```
function S'Succ(Arg : S'Base)
  return S'Base
```

39 For an enumeration type, the function returns the value whose position number is one more than that of the value of *Arg*; 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*. ♦ Constraint_Error is raised if there is no such machine number. See 3.5(21).

40 S'Val	For every discrete subtype S:
41	S'Val denotes a function with the following specification:

42
```
function S'Val(Arg :  integer)
  return S'Base
```

43 This function returns a value of the type of S whose position number equals the value of *Arg*. See 3.5.5(4).

44 S'Value	For every scalar subtype S:
45	S'Value denotes a function with the following specification:

46
```
function S'Value(Arg : String)
  return S'Base
```

47 This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces. See 3.5(51).

K. Language-Defined Pragmas -- Removed

L. Implementation-Defined Characteristics

The Ada language allows for certain machine dependences in a controlled manner. Each Ada implementation must document all implementation-defined characteristics: 1

- Capacity limitations of the implementation. See 1.1.3(3). 2
- The coded representation for the text of an AVA program. See 2.1(4). 3
- The control functions allowed in comments. See 2.1(14). 4
- The representation for an end of line. See 2.2(2). 5
- Maximum supported line length and lexical element length. See 2.2(15). 6
- Range bounds evaluated left to right. **AVA Requirement**. See 3.5(10). 7
- Check that the subprogram body has been elaborated before the evaluation of the actual parameters. **AVA Requirement**. See 3.11(11). 8
- Prefix of indexed_component evaluated before indices, which are then evaluated left to right. **AVA Requirement**. See 4.1.1(8). 9
- Implementation-defined attributes. See 4.1.4(13). 10
- Obtaining the values and the assignments in an aggregate proceeds left to right. **AVA Requirement**. See 4.3(5). 11
- Expression evaluation in a record_component_association_list occurs from left to right. **AVA Requirement**. See 4.3.1(19). 12
- Array component expressions of an aggregate are evaluated from left to right. **AVA Requirement**. See 4.3.3(22). 13
- The two operands of an expression of the form X op Y are evaluated left to right, before application of the operator. **AVA Requirement**. See 4.5(14). 14
- Optimization of integer expressions. **AVA Requirement**. See 4.5(14). 15
- For the evaluation of a membership test, the simple_expression and the range are evaluated from left to right. **AVA Requirement**. See 4.5.2(27). 16
- Parameters passed *by copy*. **AVA Requirement**. See 6.2(2). 17
- Order of copy-back and constraint checking upon subprogram return. **AVA Requirement**. See 6.4.1(17). 18
- The representation for a compilation. See 10.1(2). 19
- Any restrictions on compilations that contain multiple compilation_units. See 10.1(4). 20
- The mechanisms for creating an environment and for adding and replacing compilation units. See 10.1.4(3). 21
- The implementation must require that a compilation unit be legal before inserting it into the environment. **AVA Requirement**. See 10.1.4(6). 22
- The manner of explicitly assigning library units to a partition. See 10.2(2). 23
- The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit. See 10.2(2). 24
- The manner of designating the main subprogram of a partition. See 10.2(7). 25

26 • A library_item that is a library_unit_body is elaborated *immediately* after the library_unit which it completes. **AVA Requirement**. See 10.2(13).

27 • The order of elaboration of library_items. See 10.2(18).

28 • The mechanisms for building and running partitions. See 10.2(24).

29 • The details of program execution, including program termination. See 10.2(25).

30 • The contents of the visible part of package System and its language-defined children. See 13.7(2).

31 • The names and characteristics of the numeric subtypes declared in the visible part of package Standard. See A.1(3).

32 • Any implementation-defined characteristics of the input-output packages. See A.7(14).

33 • external files for standard input, standard output, and standard error See A.10(5).

M. Glossary

This Annex contains informal descriptions of some terms used in this Reference Manual. To find more formal definitions, look the term up in the index. 1

Ada Commentary Integration Document. The Ada Commentary Integration Document (ACID) is an edition of RM83 in which clearly marked insertions and deletions indicate the effect of integrating the approved AIs. 2

Ada Issue. An Ada Issue (AI) is a numbered ruling from the ARG. 3

Ada Rapporteur Group. The Ada Rapporteur Group (ARG) interprets the RM83. 4

Array type. An array type is a composite type whose components are all of the same type. Components are selected by indexing. 5

Character type. A character type is an enumeration type whose values include characters. 6

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 or the body ♦. 7

Composite type. A composite type has components. 8

Construct. A *construct* is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under ‘‘Syntax.’’ 9

Declaration. A *declaration* is a language construct that associates a name with (a view of) an entity. 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). 10

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 ♦ formal parameter names♦, 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)). 11

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. 12

Elementary type. An elementary type does not have components. 13

Enumeration type. An enumeration type is defined by an enumeration of its values, which may be named by identifiers or character literals. 14

Exception. An *exception* represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an *exception occurrence*. To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called *handling* the exception. 15

16 **Execution**. The process by which a construct achieves its run-time effect is called *execution*. Execution of a declaration is also called *elaboration*. Execution of an expression is also called *evaluation*.

17 **Integer type**. Integer types comprise the signed integer 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. ♦

18 **Library unit**. A library unit is a separately compiled program unit, and is always a package or a subprogram♦. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a *subsystem*.

19 **Object**. An object is either a constant or a variable. An object contains a value. An object is created by an object_declaration ♦. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

20 **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 ♦) 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. A package may also include axioms, purported theorems, and specification functions.

21 **Partition**. A *partition* is a ♦ program. ♦ A partition consists of a set of library units. ♦ A program may only contain one partition.

22 **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. ♦

23 **Private type**. A private type is a partial view of a type whose full view is hidden from its clients.

24 **Program unit**. A *program unit* is either a package♦ 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.

25 **Program**. A *program* is a ♦ partition which may execute in a separate address space ♦. A partition consists of a set of library units.

26 **Protected type**. A protected type is a composite type whose components are protected from concurrent access by multiple tasks.

27 **Record type**. A record type is a composite type consisting of zero or more named components, possibly of different types.

28 **Scalar type**. A scalar type is ♦ a discrete type♦.

29 **Subtype**. A subtype is a type together with a constraint, 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.

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 *classes*. The types of a given class share a set of primitive operations. ♦ 30

Uniformity Issue. A Uniformity Issue (UI) is a numbered recommendation from the URG. 31

Uniformity Rapporteur Group. The Uniformity Rapporteur Group (URG) issues recommendations intended to increase uniformity across Ada implementations. 32

View. (See Definition.) 33

N. Syntax Summary

This Annex summarizes the complete syntax of the language. See 1.1.4 for a description of the notation used. 1

2

2.1:
character ::= graphic_character | format_effector | other_control_function

2.1:
graphic_character ::= identifier_letter | digit | space_character | special_character

2.3:
identifier ::=
  identifier_letter {[underline] letter_or_digit}

2.3:
letter_or_digit ::= identifier_letter | digit

2.4:
numeric_literal ::= decimal_literal | based_literal

2.4.1:
decimal_literal ::= numeral ♦ [exponent]

2.4.1:
numeral ::= digit {[underline] digit}

2.4.1:
exponent ::= E + numeral | ♦

2.4.2:
based_literal ::=
  base # 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

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

2.7:
comment ::= --{*non_end_of_line*_character}

2.10:
annotation_line ::= --|{*non_end_of_line*_character}

3.1:
basic_declaration ::=

```
    type_declaration            | subtype_declaration
  | **inner_declaration**       | ♦
  | subprogram_declaration      | ♦
  | package_declaration         | renaming_declaration
  | **axiom_decl**
  | **theorem_decl**
  | **defun_decl**
  | ♦                           | ♦
  | ♦
```

3.1:
defining_identifier ::= identifier

3.1:
inner_declaration ::=
    object_declaration
  | number_declaration
  | **invariant_annotation**

3.2.1:
type_declaration ::= full_type_declaration
  | ♦
  | private_type_declaration
  | ♦

3.2.1:
full_type_declaration ::=
    type defining_identifier ♦ **is** type_definition;
  | ♦

3.2.1:
type_definition ::=
    enumeration_type_definition | ♦
  | ♦ | array_type_definition
  | record_type_definition | ♦
  | ♦

3.2.2:
subtype_declaration ::=
  subtype defining_identifier **is** subtype_indication;

3.2.2:
subtype_indication ::= 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 | ♦

3.2.2:
composite_constraint ::=
    index_constraint | ♦

3.3.1:
object_declaration ::=
   defining_identifier_list : [**constant**] subtype_indication [:= expression];
  | ♦
  | ♦
  | ♦

3.3.1:
defining_identifier_list ::= defining_identifier {, defining_identifier}

3.3.2:
number_declaration ::=
    defining_identifier_list : **constant** := *static*_expression;

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

3.6:
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 (*integer*_subtype_definition {, *integer*_subtype_definition}) **of** component_definition

3.6:
discrete_subtype_definition ::= *discrete*_subtype_*mark* | range

3.6:
integer_subtype_definition ::=
***integer*_**subtype_*mark* | range

3.6:
component_definition ::= ♦ subtype_indication

3.6.1:
index_constraint ::= (discrete_range {, discrete_range})

3.6.1:
discrete_range ::= *discrete*_subtype_indication | range

3.8:
record_type_definition ::= ♦ record_definition

3.8:
record_definition ::=
   record
     component_list
   end record
  | ♦

3.8:
component_list ::=
    component_item {component_item}
  | ♦
  | **null**;

3.8:
component_item ::= component_declaration♦

3.8:
component_declaration ::=
  defining_identifier_list : component_definition ♦;

3.11:
declarative_part ::= {declarative_item}

3.11:
declarative_item ::=
   basic_declarative_item | body

3.11:
basic_declarative_item ::=
   basic_declaration | ♦

3.11:
body ::= proper_body | ♦

3.11:
proper_body ::=
   subprogram_body | package_body | ♦

3.11:
inner_declarative_part ::= **inner_declaration**

3.12:
assert_annotation ::= **assert** logical_expression ;

3.12:
invariant_annotation ::= **invariant** logical_expression ;

3.12:
transition_annotation ::= **where** logical_expression ;

3.12:
subprogram_annotation ::=
    where logical_expression
  | **where return** [identifier ,] logical_expression

3.12:
axiom_decl ::= **axiom** identifier logical_expression ;

3.12:
theorem_decl ::= **theorem** identifier logical_expression ;

3.12:
defun_decl ::= **defun** identifier arglist logical_expression ;

3.12:
arglist ::= ({identifier})

4.1:
```
name ::=
    direct_name          | ♦
  | indexed_component    | ♦
  | selected_component   | attribute_reference
  | type_conversion      | function_call
  | character_literal
```

4.1:
direct_name ::= identifier | ♦

4.1:
prefix ::= name | ♦

4.1.1:
indexed_component ::= prefix(expression {, expression})

4.1.3:
selected_component ::= prefix . selector_name

4.1.3:
selector_name ::= identifier | ♦

4.1.4:
attribute_reference ::= prefix’attribute_designator

4.1.4:
attribute_designator ::=
   identifier[(*static*_expression)]
  | ♦

4.1.4:
range_attribute_reference ::= prefix’range_attribute_designator

4.1.4:
range_attribute_designator ::= Range[(*static*_expression)]

4.3:
aggregate ::= record_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}
  | ♦

4.3.1:
record_component_association ::=
  [component_choice_list =>] expression

4.3.1:
component_choice_list ::=
    *component*_selector_name {| *component*_selector_name}
  | **others**

4.3.3:
array_aggregate ::=
 positional_array_aggregate | named_array_aggregate

4.3.3:
positional_array_aggregate ::=
    (expression, expression {, expression})
  | ♦

4.3.3:
named_array_aggregate ::=
   (**others** => **expression**)

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 | ♦ | string_literal | aggregate
 | name | qualified_expression | ♦ | (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)
  | ♦

4.7:
qualified_expression ::=
  subtype_mark’(expression) | subtype_mark’aggregate

4.10:
```
logical_expression ::=
    expression
  | env_expression
  | if logical_expression
      then expression
      else expression
    fi
  | logical_expression iff logical_expression
  | logical_expression implies logical_expression
  | all identifier [ in logical_expression ] , logical_expression
```

4.10:
```
env_expression ::=
    @ identifier
  | in expression
  | out expression
```

5.1:
```
sequence_of_statements ::= statement {statement}
```

5.1:
```
statement ::=
  ♦ simple_statement | ♦ ava_compound_statement
```

5.1:
```
simple_statement ::=
    null_statement           | assert_annotation
  | assignment_statement     | exit_statement
  | ♦                        | procedure_call_statement
  | return_statement         | ♦
  | ♦                        | ♦
  | ♦                        | raise_statement
  | ♦
```

5.1:
```
compound_statement ::=
    if_statement             | case_statement
  | loop_statement           | block_statement
```

5.1:
```
ava_compound_statement ::=
 [ logical_annotation ] compound_statement
```

5.1:
```
null_statement ::= null;
```

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

5.4:
discrete_choice_list ::= discrete_choice {| discrete_choice}

5.4:
discrete_choice ::= expression | discrete_range | **others**

5.5:
loop_statement ::=
  ♦
    [iteration_scheme] **loop**
      sequence_of_statements
    end loop ♦;

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 ::=
  ♦
    [**declare**
      inner_part]
    begin
      handled_sequence_of_statements
    end ♦;

5.7:
exit_statement ::=
  exit ♦ ;

6.1:
subprogram_declaration ::=
  subprogram_specification; [**subprogram_annotation**;]

6.1:
subprogram_specification ::=
  procedure defining_program_unit_name parameter_profile
  | **function** defining_designator parameter_and_result_profile

6.1:
designator ::= [parent_unit_name .] identifier | ♦

6.1:
defining_designator ::= defining_program_unit_name | ♦

6.1:
defining_program_unit_name ::= [parent_unit_name .] defining_identifier

6.1:
parameter_profile ::= [formal_part]

6.1:
parameter_and_result_profile ::= [formal_part] **return** subtype_mark

6.1:
formal_part ::=
  (parameter_specification {; parameter_specification})

6.1:
parameter_specification ::=
  defining_identifier_list : mode subtype_mark ♦
  | ♦

6.1:
mode ::= [**in**] | **in out** | ♦

6.3:
subprogram_body ::=
  subprogram_specification **is**
    inner_declarative_part
  begin
    handled_sequence_of_statements
  end [designator];
  [**subprogram_annotation**;]

6.4:
procedure_call_statement ::= *procedure_***name** [actual_parameter_part] ;

6.4:
function_call ::= ♦ | *function*_prefix actual_parameter_part

6.4:
actual_parameter_part ::= (parameter_association {, parameter_association})

6.4:
parameter_association ::= ♦ explicit_actual_parameter

6.4:
explicit_actual_parameter ::= expression | *variable*_name

6.5:
return_statement ::= **return** [expression];

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 **is private**;

8.4:
use_clause ::= use_package_clause | ♦

8.4:
use_package_clause ::= **use** *package*_name {, *package*_name};

8.5:
renaming_declaration ::=
    object_renaming_declaration
  | ♦
  | package_renaming_declaration
  | subprogram_renaming_declaration
  | ♦

8.5.1:
object_renaming_declaration ::= defining_identifier : subtype_mark **renames** *object*_name;

8.5.3:
package_renaming_declaration ::= **package** defining_program_unit_name **renames** *package*_name;

8.5.4:
subprogram_renaming_declaration ::= subprogram_specification **renames** *callable_entity*_name;

10.1.1:
compilation ::= {compilation_unit}

10.1.1:
compilation_unit ::=
   context_clause library_item
  | ♦

10.1.1:
library_item ::= ♦ library_unit_declaration
  | library_unit_body
  | ♦

10.1.1:
library_unit_declaration ::=
    subprogram_declaration | package_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 ::= **with** *library_unit_*name {, *library_unit_*name};

11.2:
handled_sequence_of_statements ::=
    sequence_of_statements
  [**exception**
    exception_handler
   ♦]

11.2:
exception_handler ::=
  when ♦ exception_choice ♦ =>
    sequence_of_statements

11.2:
exception_choice ::= ♦ **others**

11.3:
raise_statement ::= **raise** Program_Error;

Syntax Cross Reference

_subtype_definition		
	constrained_array_definition	3.6
actual_parameter_part		
	function_call	6.4
	procedure_call_statement	6.4
aggregate		
	primary	4.4
	qualified_expression	4.7
arglist		
	defun_decl	3.12
array_aggregate		
	aggregate	4.3
array_type_definition		
	type_definition	3.2.1
assert_annotation		
	simple_statement	5.1
assignment_statement		
	simple_statement	5.1
attribute_designator		
	attribute_reference	4.1.4
attribute_reference		
	name	4.1
axiom_decl		
	basic_declaration	3.1
base		
	based_literal	2.4.2
based_literal		
	numeric_literal	2.4
based_numeral		
	based_literal	2.4.2
basic_declaration		
	basic_declarative_item	3.11
basic_declarative_item		
	declarative_item	3.11
	package_specification	7.1
binary_adding_operator		
	simple_expression	4.4
block_statement		
	compound_statement	5.1
body		
	declarative_item	3.11
case_statement		
	compound_statement	5.1
case_statement_alternative		
	case_statement	5.4
character		
	annotation_line	2.10
	comment	2.7
character_literal		
	defining_character_literal	3.5.1
	name	4.1
compilation_unit		
	compilation	10.1.1
component_choice_list		
	record_component_association	4.3.1
component_declaration		
	component_item	3.8
component_definition		
	component_declaration	3.8
	constrained_array_definition	3.6
	unconstrained_array_definition	3.6
component_item		
	component_list	3.8
component_list		
	record_definition	3.8
composite_constraint		
	constraint	3.2.2
condition		
	if_statement	5.3
	iteration_scheme	5.5
constrained_array_definition		
	array_type_definition	3.6
constraint		
	subtype_indication	3.2.2
context_clause		
	compilation_unit	10.1.1
context_item		
	context_clause	10.1.2
decimal_literal		
	numeric_literal	2.4
declarative_item		
	declarative_part	3.11
declarative_part		
	package_body	7.2
defining_character_literal		
	enumeration_literal_specification	3.5.1
defining_designator		
	subprogram_specification	6.1
defining_identifier		
	defining_identifier_list	3.3.1
	defining_program_unit_name	6.1
	enumeration_literal_specification	3.5.1

full_type_declaration	3.2.1
loop_parameter_specification	5.5
object_renaming_declaration	8.5.1
private_type_declaration	7.3
subtype_declaration	3.2.2
defining_identifier_list	
component_declaration	3.8
number_declaration	3.3.2
object_declaration	3.3.1
parameter_specification	6.1
defining_program_unit_name	
defining_designator	6.1
package_body	7.2
package_renaming_declaration	8.5.3
package_specification	7.1
subprogram_specification	6.1
defun_decl	
basic_declaration	3.1
designator	
subprogram_body	6.3
digit	
extended_digit	2.4.2
graphic_character	2.1
letter_or_digit	2.3
numeral	2.4.1
direct_name	
name	4.1
discrete_choice	
discrete_choice_list	5.4
discrete_choice_list	
case_statement_alternative	5.4
discrete_range	
discrete_choice	5.4
index_constraint	3.6.1
discrete_subtype_definition	
loop_parameter_specification	5.5
enumeration_literal_specification	
enumeration_type_definition	3.5.1
enumeration_type_definition	
type_definition	3.2.1
env_expression	
logical_expression	4.10
exception_choice	
exception_handler	11.2
exception_handler	
handled_sequence_of_statements	11.2
exit_statement	
simple_statement	5.1
explicit_actual_parameter	
parameter_association	6.4
exponent	
based_literal	2.4.2
decimal_literal	2.4.1
expression	
assignment_statement	5.2
attribute_designator	4.1.4
case_statement	5.4
condition	5.3
discrete_choice	5.4
env_expression	4.10
explicit_actual_parameter	6.4
indexed_component	4.1.1
logical_expression	4.10
number_declaration	3.3.2
object_declaration	3.3.1
positional_array_aggregate	4.3.3
primary	4.4
qualified_expression	4.7
range_attribute_designator	4.1.4
record_component_association	4.3.1
return_statement	6.5
type_conversion	4.6
extended_digit	
based_numeral	2.4.2
factor	
term	4.4
formal_part	
parameter_and_result_profile	6.1
parameter_profile	6.1
format_effector	
character	2.1
full_type_declaration	
type_declaration	3.2.1
function_call	
name	4.1
graphic_character	
character	2.1
character_literal	2.5
string_element	2.6
handled_sequence_of_statements	
block_statement	5.6
package_body	7.2
subprogram_body	6.3
identifier	
arglist	3.12
attribute_designator	4.1.4
axiom_decl	3.12
defining_identifier	3.1
defun_decl	3.12
designator	6.1
direct_name	4.1
env_expression	4.10
logical_expression	4.10
package_body	7.2
package_specification	7.1
selector_name	4.1.3
subprogram_annotation	3.12
theorem_decl	3.12
identifier_letter	

graphic_character	2.1
identifier	2.3
letter_or_digit	2.3
if_statement	
compound_statement	5.1
index_constraint	
composite_constraint	3.2.2
index_subtype_definition	
unconstrained_array_definition	3.6
indexed_component	
name	4.1
inner_declaration	
basic_declaration	3.1
inner_declarative_part	
subprogram_body	6.3
inner_part	
block_statement	5.6
iteration_scheme	
loop_statement	5.5
letter_or_digit	
identifier	2.3
library_item	
compilation_unit	10.1.1
library_unit_body	
library_item	10.1.1
library_unit_declaration	
library_item	10.1.1
logical_expression	
assert_annotation	3.12
axiom_decl	3.12
defun_decl	3.12
invariant_annotation	3.12
logical_expression	4.10
subprogram_annotation	3.12
theorem_decl	3.12
transition_annotation	3.12
loop_parameter_specification	
iteration_scheme	5.5
loop_statement	
compound_statement	5.1
mode	
parameter_specification	6.1
multiplying_operator	
term	4.4
name	
assignment_statement	5.2
explicit_actual_parameter	6.4
object_renaming_declaration	8.5.1
package_renaming_declaration	8.5.3
parent_unit_name	10.1.1
prefix	4.1
primary	4.4
procedure_call_statement	6.4
raise_statement	11.3
subprogram_renaming_declaration	8.5.4
subtype_mark	3.2.2
use_package_clause	8.4
with_clause	10.1.2
named_array_aggregate	
array_aggregate	4.3.3
null_statement	
simple_statement	5.1
numeral	
base	2.4.2
decimal_literal	2.4.1
exponent	2.4.1
numeric_literal	
primary	4.4
object_renaming_declaration	
renaming_declaration	8.5
other_control_function	
character	2.1
package_body	
library_unit_body	10.1.1
proper_body	3.11
package_declaration	
basic_declaration	3.1
library_unit_declaration	10.1.1
package_renaming_declaration	
renaming_declaration	8.5
package_specification	
package_declaration	7.1
parameter_and_result_profile	
subprogram_specification	6.1
parameter_association	
actual_parameter_part	6.4
parameter_profile	
subprogram_specification	6.1
parameter_specification	
formal_part	6.1
parent_unit_name	
defining_program_unit_name	6.1
designator	6.1
package_body	7.2
package_specification	7.1
positional_array_aggregate	
array_aggregate	4.3.3
prefix	
attribute_reference	4.1.4
function_call	6.4
indexed_component	4.1.1
range_attribute_reference	4.1.4
selected_component	4.1.3

primary		
	factor	4.4
private_type_declaration		
	type_declaration	3.2.1
procedure_call_statement		
	simple_statement	5.1
proper_body		
	body	3.11
qualified_expression		
	primary	4.4
raise_statement		
	simple_statement	5.1
range		
	discrete_range	3.6.1
	discrete_subtype_definition	3.6
	range_constraint	3.5
	relation	4.4
range_attribute_designator		
	range_attribute_reference	4.1.4
range_attribute_reference		
	range	3.5
range_constraint		
	scalar_constraint	3.2.2
record_aggregate		
	aggregate	4.3
record_component_association		
	record_component_association_list	4.3.1
record_component_association_list		
	record_aggregate	4.3.1
record_definition		
	record_type_definition	3.8
record_type_definition		
	type_definition	3.2.1
relation		
	expression	4.4
relational_operator		
	relation	4.4
renaming_declaration		
	basic_declaration	3.1
return_statement		
	simple_statement	5.1
scalar_constraint		
	constraint	3.2.2
selected_component		
	name	4.1
selector_name		
	component_choice_list	4.3.1
	selected_component	4.1.3
sequence_of_statements		
	case_statement_alternative	5.4
	exception_handler	11.2
	handled_sequence_of_statements	11.2
	if_statement	5.3
	loop_statement	5.5
simple_expression		
	range	3.5
	relation	4.4
space_character		
	graphic_character	2.1
special_character		
	graphic_character	2.1
statement		
	sequence_of_statements	5.1
string_element		
	string_literal	2.6
string_literal		
	primary	4.4
subprogram_annotation		
	subprogram_body	6.3
	subprogram_declaration	6.1
subprogram_body		
	library_unit_body	10.1.1
	proper_body	3.11
subprogram_declaration		
	basic_declaration	3.1
	library_unit_declaration	10.1.1
subprogram_renaming_declaration		
	renaming_declaration	8.5
subprogram_specification		
	subprogram_body	6.3
	subprogram_declaration	6.1
	subprogram_renaming_declaration	8.5.4
subtype_		
	discrete_subtype_definition	3.6
subtype_declaration		
	basic_declaration	3.1
subtype_indication		
	component_definition	3.6
	discrete_range	3.6.1
	object_declaration	3.3.1
	subtype_declaration	3.2.2
subtype_mark		
	index_subtype_definition	3.6
	object_renaming_declaration	8.5.1
	parameter_and_result_profile	6.1
	parameter_specification	6.1
	qualified_expression	4.7
	relation	4.4
	subtype_indication	3.2.2
	type_conversion	4.6
term		

simple_expression	4.4
theorem_decl	
basic_declaration	3.1
type_conversion	
name	4.1
type_declaration	
basic_declaration	3.1
type_definition	
full_type_declaration	3.2.1
unary_adding_operator	
simple_expression	4.4
unconstrained_array_definition	
array_type_definition	3.6
underline	
based_numeral	2.4.2
identifier	2.3
numeral	2.4.1
use_clause	
context_item	10.1.2
use_package_clause	
use_clause	8.4
with_clause	
context_item	10.1.2

References

[ARTEWG 87] ACM Special Interest Group on Ada, Runtime Environment Working Group.
Catalogue of Ada Runtime Implementation Dependencies.
ACM, 1987.

[Carre 88] B. A. Carre and T. J. Jennings.
SPARK - The SPADE Ada Kernel (Version 1.0).
Technical Report, University of Southampton, March, 1988.

[Courant 83] Ada Project.
Ada/Ed Semantic Actions (Version 1.1).
Technical Report, Courant Institute, New York University, 1983.

[Courant 84] Ada Project.
Executable Semantic Model for Ada (Version 1.4).
Technical Report, Courant Institute, New York University, 1984.

[DDC 87] *The Draft Formal Definition of Ada*
Denmark, 1987.

[DoD 83] *Reference Manual for the Ada Programming Language*
United States Department of Defense, 1983.
ANSI/MIL-STD-1815 A.

[ISO 94] *Annotated Ada Reference Manual, Language and Standard Libraries, Version 6.0*
ISO/IEC, Cambridge, Massachusetts, 1994.
ISO/IEC JTC1/SC22 WG9 N 193.

[Kaufmann 94] M.J. Kaufmann, J S. Moore.
Design Goals of ACL2.
Technical Report 101, Computational Logic, Inc., August, 1994.

[Luckham 90] D. Kapur (editor).
Texts and Monographs in Computer Science: *Programming With Specifications. An Introduction to ANNA, A Language for Annotating Ada Programs.*
Springer-Verlag, 1990.

[Marsh 94] William Marsh.
Formal Semantics of SPARK, Static Semantics.
Technical Report, Program Validation Ltd., October, 1994.

[O’Neill 94] Ian O’Neill.
Formal Semantics of SPARK, Dynamic Semantics.
Technical Report, Program Validation Ltd., October, 1994.

[Polak 88] Wolfgang Polak.
A Technique for Defining Predicate Transformers.
Technical Report 17-4, Odyssey Research Associates, Ithaca, NY, October, 1988.

[Ramsey 88] Norman Ramsey.
Developing Formally Verified Ada Programs.
Technical Report 17-3, Odyssey Research Associates, Ithaca, NY, October, 1988.

[Smith 95] M.K. Smith.
Dynamic Formal Semantics for AVA 95.
Technical Report 112, Computational Logic, Inc., September, 1995.

Index