The AVA Reference Manual: Derived from ANSI/MIL-STD-1815A-1983

Modifications by
Michael K. Smith

Technical Report: 64 February 1992

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

This document is derived from the on-line version of the Ada Reference Manual (ANSI/MIL-STD-1815A-1983) available at ajpo.sei.cmu.edu.

This work was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views, conclusions and modifications contained in this document are those of the authors and should not be interpreted as representing the official policies either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

The AVA Reference Manual

This manual consists of the text of ANSI/MIL-STD-1815A-1983, "Reference Manual for the Ada Programming Language", with modifications and annotations by Computational Logic, Inc. The form of the modifications and annotations is described in *Forward to the AVA Revision*.

Computational Logic, Inc.
1717 W. 6th Street, Suite 290
Austin, TX 78703-7441

Library of Congress Cataloging-in-Publication Data
Smith, Michael K., Editor, 1949-
The AVA Reference Manual
ISBN
1. Programming Languages - Ada
LCC Number:

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Foreword to the AVA revision

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 [DoD 83] and represents the *informal* description of AVA. Work on the formal definition is ongoing.

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 first 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.

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 *omitted* or *removed*. 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. In some cases this additional text was motivated by a binding Ada Interpretation (AI), in which case it is labeled by a form like "[AI-0001]". We have also added to each section a list of cross-references to associated AI’s as of January 1988. Deletions in *References*, the appendices, and the the AI cross-references have *not* been scrupulously tracked.

This document is based on the on-line version of the *Reference Manual for the Ada Programming Language* [DoD 83] (hereafter ARM) available at ajpo.sei.cmu.com. This modified version was created by Michael K. Smith of Computational Logic. Substantial discussions on the details of restrictions were carried out with Dan Craigen and Mark Saaltink (now of Odyssey Research Associates). They also reviewed a draft of this document and detected numerous errors. Many of the detailed modifications were inspired by the extensive discussions available in the accumulated Ada Interpretations.

Predictability and critical systems

Computational Logic 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.

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.

Other work

There have been two motivations for work on Ada subsets.

(a) To get a dialect with predictable behavior for safety and security critical systems.

(b) To carve out a subset for which a reasonably tractable formal definition can be provided.

The second is ultimately in support of the first.

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

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.

Odyssey Research Associates is working on the "Penelope System" [Ramsey 88, Polak 88]. It has developed a formal definition for a language that corresponds to a substantial subset of Ada. Its formally defined language has a more regular semantics than a literal Ada definition would.

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] (ARM). Conformance to the complete ARM presents some unsolvable problems. The EEC definition was unable to define parts of the language because the definition embodied in the ARM 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.

We have two observations with regard to the EEC definition.

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

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

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 Program Validation Ltd.

Wichmann has proposed a lower level language, called Low-Ada [Wichmann 89a], designed to avoid insecurities in the full Ada language [Wichmann 89b].

> This language is not an ordinary programming language but is an intermediate form for representing Ada programs to facilitate the production of high integrity software using Ada.

The intent is that compilers will be capable of generating Low-Ada, which would then be the input for validation tools.

Foreword to the Original Ada LRM

Ada is the result of a collective effort to design a common language for programming large scale and real-time systems.

The common high order language program began in 1974. The requirements of the United States Department of Defense were formalized in a series of documents which were extensively reviewed by the Services, industrial organizations, universities, and foreign military departments. The Ada language was designed in accordance with the final (1978) form of these requirements, embodied in the Steelman specification.

The Ada design team was led by Jean D. Ichbiah and has included Bernd Krieg-Brueckner, Brian A. Wichmann, Henry F. Ledgard, Jean-Claude Heliard, Jean-Loup Gailly, Jean-Raymond Abrial, John G.P. Barnes, Mike Woodger, Olivier Roubine, Paul N. Hilfinger, and Robert Firth.

At various stages of the project, several people closely associated with the design team made major contributions. They include J.B. Goodenough, R.F. Brender, M.W. Davis, G. Ferran, K. Lester, L. MacLaren, E. Morel, I.R. Nassi, I.C. Pyle, S.A. Schuman, and S.C. Vestal.

Two parallel efforts that were started in the second phase of this design had a deep influence on the language. One was the development of a formal definition using denotational semantics, with the participation of V. Donzeau-Gouge, G. Kahn, and B. Lang. The other was the design of a test translator with the participation of K. Ripken, P. Boullier, P. Cadiou, J. Holden, J.F. Hueras, R.G. Lange, and D.T. Cornhill. The entire effort benefitted from the dedicated assistance of Lyn Churchill and Marion Myers, and the effective technical support of B. Gravem, W.L. Heimerdinger, and P. Cleve. H.G. Schmitz served as program manager.

Over the five years spent on this project, several intense week-long design reviews were conducted, with the participation of P. Belmont, B. Brosgol, P. Cohen, R. Dewar, A. Evans, G. Fisher, H. Harte, A.L. Hisgen, P. Knueven, M. Kronental, N. Lomuto, E. Ploedereder, G. Seegmueller, V. Stenning, D. Taffs, and also F. Belz, R. Converse, K. Correll, A.N. Habermann, J. Sammet, S. Squires, J. Teller, P. Wegner, and P.R. Wetherall.

Several persons had a constructive influence with their comments, criticisms and suggestions. They include P. Brinch Hansen, G. Goos, C.A.R. Hoare, Mark Rain, W.A. Wulf, and also E. Boebert, P. Bonnard, H. Clausen, M. Cox, G. Dismukes, R. Eachus, T. Froggatt, H. Ganzinger, C. Hewitt, S. Kamin, R. Kotler, O. Lecarme, J.A.N. Lee, J.L. Mansion, F. Minel, T. Phinney, J. Roehrich, V. Schneider, A. Singer, D. Slosberg, I.C. Wand, the reviewers of Ada-Europe, AdaTech, Afcet, those of the LMSC review team, and those of the Ada Tokyo Study Group.

These reviews and comments, the numerous evaluation reports received at the end of the first and second phase, the nine hundred language issue reports and test and evaluation reports received from fifteen different countries during the third phase of the project, the thousands of comments received during the Ansi Canvass, and the on-going work of the Ifip Working Group 2.4 on system implementation languages and that of the Purdue Europe LTPL-E committee, all had a substantial influence on the final definition of Ada.

The Military Departments and Agencies have provided a broad base of support including funding, extensive reviews, and countless individual contributions by the members of the High Order Language

Working Group and other interested personnel. In particular, William A. Whitaker provided leadership for the program during the formative stages. David A. Fisher was responsible for the successful development and refinement of the language requirement documents that led to the Steelman specification.

This language definition was developed by Cii Honeywell Bull and later Alsys, and by Honeywell Systems and Research Center, under contract to the United States Department of Defense. William E. Carlson and later Larry E. Druffel served as the technical representatives of the United States Government and effectively coordinated the efforts of all participants in the Ada program.

Chapter 1
INTRODUCTION

AVA (A Verifiable Ada) is a subset of Ada. 1

1.1 Scope of the Standard

This standard specifies the form and meaning of program units written in the AVA subset of Ada. Its purpose is to promote formal specification and proofs of correctness of programs written in this subset. 1

1.1.1 Extent of the Standard

This standard specifies: 1

(a) The form of a program unit written in AVA. 2

(b) The effect of translating and executing such a program unit. 3

(c) The manner in which program units may be combined to form AVA programs. 4

(d) The predefined program units that a conforming implementation must supply. 5

(e) The permissible variations within the standard, and the manner in which they must be specified. 6

(f) Those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program unit containing such violations. 7

(g) Omitted 8

This standard does not specify: 9

(h) The means whereby a program unit written in AVA is transformed into object code executable by a processor. 10

(i) The means whereby translation or execution of program units is invoked and the executing units are controlled. 11

(j) The size or speed of the object code, or the relative execution speed of different language constructs. 12

(k) The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages. 13

(l) Omitted 14

(m) The size of a program or program unit that will exceed the capacity of a particular conforming implementation. 15

Where this standard specifies that a program unit written in AVA has an exact effect, this effect is the operational meaning of the program unit and must be produced by all conforming implementations. Where this standard specifies permissible variations in the effects of constituents of a program unit written in AVA, the operational meaning of the program unit as a whole is understood to be the range of possible effects that result from all these variations, and a conforming implementation is allowed to produce any of these possible effects. ♦ 16

In some places this standard requires more specific behavior from a conforming implementation than Ada does. For example, AVA specifies a left to right order of evaluation for the actual parameters to a subprogram call. Such places are marked with **IMPLEMENTATION REQUIREMENT**. See section 1.6. 16a

1.1.2 Conformity of an Implementation With the Standard

A conforming implementation is one that: 1

(a) Correctly translates and executes legal program units written in AVA, provided that they are not so large as to exceed the capacity of the implementation. 2

(b) Rejects all program units that are so large as to exceed the capacity of the implementation. 3

(c) Rejects all program units that contain errors whose detection is required by the standard. 4

(d) Supplies all predefined program units required by the standard. 5

(e) Contains no variations except where the standard permits. 6

(f) Specifies all such permitted variations in the manner prescribed by the standard. 7

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
1.01.02	(00)	ra	WJ	0325/05	86-07-23	Implementation-dependent limitations

1.2 Structure of the Standard

This reference manual contains various chapters, annexes, appendices, and an index. They are numbered to conform to the corresponding sections of the ARM. 1

Each chapter is divided into sections that have a common structure. Each section introduces its subject, gives any necessary syntax rules, and describes the semantics of the corresponding language constructs. Examples and notes, and then references, may appear at the end of a section. 2

Examples are meant to illustrate the possible forms of the constructs described. Notes are meant to emphasize consequences of the rules described in the section or elsewhere. References are meant to attract the attention of readers to a term or phrase having a technical meaning defined in another section. 3

The informal standard definition of the AVA programming language consists of the ♦ chapters and the ♦ annexes, subject to the following restriction: the material in each of the items listed below is informative, and not part of the standard definition of the AVA programming language: 4

- Omitted 5
- Section 1.4 Language summary 6
- The examples, notes, and references given at the end of each section 7

- Each section whose title starts with the word "Example" or "Examples" 8

1.3 Design Goals and Sources: Omitted

1.4 Language Summary

An AVA program is composed of one or more program units. These program units can be compiled separately. Program units may be subprograms (which define executable algorithms) ♦ or package units (which define collections of entities). Each 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. 1

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. 2

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. The text of a separately compiled program unit must name the library units it requires. 3

Program Units 4

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. 5

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. 6

A package is the basic unit for defining a collection of logically related entities. For example, a package can be used to define a common pool of data and types, a collection of related subprograms, or 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. 7

♦ 8

Declarations and Statements 9

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. 10

The declarative part associates names with declared entities. For example, a name may denote ♦ a constant, or a variable. A package declarative part also introduces ♦ types and the names and parameters of other nested subprograms and packages to be used in the package unit. 11

The sequence of statements describes a sequence of actions that are to be performed. The statements are executed in succession (unless an exit or return statement, or the raising of an exception, causes execution to continue from another place). 12

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. 13

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. 14

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. 15

A block statement comprises a sequence of statements preceded by the declaration of local entities used by the statements. 16

♦ 17

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. 18

Data Types 19

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 scalar types (comprising enumeration and numeric types), composite types, ♦ and private types. 20

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. 21

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. 22

Composite types allow definitions of structured objects with related components. The composite types in the language provide for 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. 23

♦ 24

Private types can be defined in a package that conceals structural details that are externally irrelevant. Only the logically necessary properties ♦ are made visible to the users of such types. 26

The concept of a type is 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 ♦. 27

Other Facilities 28

♦ 29

Input-output is defined in the language by means of predefined library packages. Facilities are provided for input-output of characters and strings. 30

♦ 31

1.5 Method of Description and Syntax Notation

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

The informal meaning[1] of AVA program units is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs. This narrative employs technical terms whose precise definition is given in the text (references to the section containing the definition of a technical term appear at the end of each section that uses the term). 2

All other terms are in the English language and bear their natural meaning, as defined in Webster's Third New International Dictionary of the English Language. 3

The context-free syntax of the language is described using a simple variant of Backus-Naur Form. In particular, 4

(a) Lower case words, some containing embedded underlines, are used to denote syntactic categories, for example: 5

adding_operator

Whenever the name of a syntactic category is used apart from the syntax rules themselves, spaces take the place of the underlines (thus: adding operator). 6

(b) Boldface words are used to denote reserved words, for example: 7

array

(c) Square brackets enclose optional items. Thus the two following rules are equivalent. 8

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

(d) Braces 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. 9

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

(e) A vertical bar separates alternative items unless it occurs immediately after an opening brace, in which case it stands for itself: 10

letter_or_digit ::= letter | digit
component_association ::= [choice {| choice} =>] expression

(f) 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 11

[1] See [SmithMK 88, SmithMK 90] for descriptions of progress toward a formal definition of AVA.

information. For example *type*_name ♦ is equivalent to name alone.

Note:

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 12

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

Different lines are used for parts of a syntax rule if the corresponding parts of the construct described by the rule are intended to be on different lines. Indentation in the rule is a recommendation for indentation of the corresponding part of the construct. It is recommended that all indentations be by multiples of a basic step of indentation (the number of spaces for the basic step is not defined). The preferred places for other line breaks are after semicolons. On the other hand, if a complete construct can fit on one line, this is also allowed in the recommended paragraphing. 13

1.6 Classification of Errors

The language definition classifies errors into several different categories: 1

(a) Errors that must be detected at compilation time by every AVA compiler. 2

These errors correspond to any violation of a rule given in this reference manual, other than the violations that correspond to (b) below ♦. In particular, violation of any rule that uses the terms *must*, *allowed*, *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. 3

(b) Errors that must be detected at run time by the execution of an AVA program. 4

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 an exception is certain to be raised in every execution of a program, then compilers are allowed (although not required) to report this fact at compilation time. 5

(c) Erroneous execution. Omitted. 6

(d) Incorrect order dependences. Omitted. 8

Note the elimination of erroneous execution and incorrect order dependencies. This has been accomplished in two ways. 9a

(a) We have removed or restricted some constructs that permit the kinds of ambiguity that lead to erroneous behavior. 9b

(b) We have specified *one* of the allowed Ada behaviors to be the allowed AVA behavior. These are marked with **IMPLEMENTATION REQUIREMENT** in the text. 9c

Order of evaluation in all important cases (e.g. state changing cases) is now specified. This 9d

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. It is certainly better than 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.[2] 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 ''poor style''. In general, these correspond to practices that can lead to compiler dependent behavior, even when executing code compiled by conforming Ada compilers.

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
1.06	(02)	BI	WA	0256/23	88-06-16	"Successful" compilation
1.06	(03)	BI	RE	0613/00	88-12-21	Invalid pragmas are not illegal
1.06	(09)	BI	WI	0284/01	87-06-09	Definition of incorrect order dependence
1.06	(10)	ra	RE	0579/00	88-08-31	Compile time detection of erroneous constructs

[2]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.

Chapter 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 chapter. ♦ 1

References: character 2.1, compilation 10.1, lexical element 2.2 ♦ 2

2.1 Character Set

The only characters allowed in the text of a program are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the *ISO* seven-bit coded character set (*ISO* standard 646), and is represented (visually) by a graphical symbol. Some graphic characters are represented by different graphical symbols in alternative national representations of the *ISO* character set. The description of the language definition in this standard reference manual uses the *ASCII* graphical symbols, the *ANSI* graphical representation of the *ISO* character set. 1

```
graphic_character ::= basic_graphic_character
  | lower_case_letter | other_special_character

basic_graphic_character ::=
    upper_case_letter | digit
  | special_character | space_character

basic_character ::=
    basic_graphic_character | format_effector
```
2

The basic character set is sufficient for writing any program. The characters included in each of the categories of basic graphic characters are defined as follows: 3

(a) upper case letters 4
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(b) digits 5
0 1 2 3 4 5 6 7 8 9

(c) special characters 6
" # & ’ ( ) * + , - . / : ; < = > _ |

(d) the space character 7

Format effectors are the *ISO* (and *ASCII*) characters called horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed. 8

The characters included in each of the remaining categories of graphic characters are defined as follows: 9

(e) lower case letters 10
a b c d e f g h i j k l m n o p q r s t u v w x y z

(f) other special characters 11
! $ % ? @ [\ ] ^ ‘ { } ~

Allowable replacements for the special characters vertical bar (|), sharp (#), and quotation (") are defined in section 2.10. 12

Notes:

The *ISO* character that corresponds to the sharp graphical symbol in the *ASCII* representation appears as a pound sterling symbol in the French, German, and United Kingdom standard national representations. In any case, the font design of graphical symbols (for example, whether they are in italic or bold typeface) is not part of the *ISO* standard. 13

The meanings of the acronyms used in this section are as follows: *ANSI* stands for American National Standards Institute, *ASCII* stands for American Standard Code for Information Interchange, and *ISO* stands for International Organization for Standardization. 14

The following names are used when referring to special characters and other special characters: 15

symbol	*name*	*symbol*	*name*
"	quotation	>	greater than
#	sharp	_	underline
&	ampersand	|	vertical bar
’	apostrophe	!	exclamation mark
(	left parenthesis	$	dollar
)	right parenthesis	%	percent
*	star, multiply	?	question mark
+	plus	@	commercial at
,	comma	[	left square bracket
-	hyphen, minus	\	back-slash
.	dot, point, period	]	right square bracket
/	slash, divide	^	circumflex
:	colon	‘	grave accent
;	semicolon	{	left brace
<	less than	}	right brace
=	equal	~	tilde

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.01	(01)	BI	WJ	0339/04	86-12-01	Allow non-English characters in comments
2.01	(01)	ST	RE	0420/03	88-11-08	Allow 256 values for type CHARACTER
2.01	(11)	ST	RE	0510/00	87-01-13	Use of national symbols and standards in an ISO standard

2.2 Lexical Elements, Separators, and Delimiters

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 either a delimiter, an identifier (which may be a reserved word), a numeric literal, a character literal, a string literal, or a comment. The effect of a program depends only on the particular sequences of lexical elements that form its compilations, excluding the comments, if any. 1

In some cases an explicit *separator* is required to separate adjacent lexical elements (namely, when without separation, interpretation as a single lexical element is possible). A separator is any of a space character, a format effector, or the end of a line. A space character is a separator except within a comment, a string literal, or a space character literal. Format effectors other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment. 2

The end of a line is always a separator. The language does not define what causes the end of a line. However if, for a given implementation, the end of a line is signified by one or more characters, then these characters must be format effectors other than horizontal tabulation. In any case, a sequence of one or more format effectors other than horizontal tabulation must cause at least one end of line. 3

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 or a numeric literal and an adjacent identifier or numeric literal. 4

A *delimiter* is either one of the following special characters (in the basic character set) 5

& ’ ( ) * + , - . / : ; < = > |

or one of the following *compound delimiters* each composed of two adjacent special characters 6

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

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. 7

The remaining forms of lexical element are described in other sections of this chapter. 8

Notes:

Each lexical element must fit on one line, since the end of a line is a separator. The quotation, sharp, and underline characters, likewise two adjacent hyphens, are not delimiters, but may form part of other lexical elements. 9

The following names are used when referring to compound delimiters: 10

delimiter	*name*
=>	arrow
..	double dot
**	double star, exponentiate
:=	assignment (pronounced: "becomes")
/=	inequality (pronounced: "not equal")
>=	greater than or equal

<=	less than or equal
<<	left label bracket
>>	right label bracket
<>	box

References: character literal 2.5, comment 2.7, compilation 10.1, format effector 2.1, identifier 2.3, numeric literal 2.4, reserved word 2.9, space character 2.1, special character 2.1, string literal 2.6 11

2.3 Identifiers

Identifiers are used as names and also as reserved words. 1

identifier ::= 2
  letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

All characters of an identifier are significant, including any underline character inserted between a letter or digit and an adjacent letter or digit. Identifiers differing only in the use of corresponding upper and lower case letters are considered as the same. 3

Examples: 4

COUNT	X	get_symbol	Ethelyn	Marion
SNOBOL_4	X1	PageCount	STORE_NEXT_ITEM	

Note:

No space is allowed within an identifier since a space is a separator. 5

References: digit 2.1, lower case letter 2.1, name 4.1, reserved word 2.9, separator 2.2, space character 2.1, upper case letter 2.1 6

2.4 Numeric Literals

There is one class of numeric literals: ♦ integer literals. ♦ Integer literals are the literals of the type *universal_integer*. 1

numeric_literal ::= decimal_literal | based_literal 2

References: literal 4.2, universal_integer type 3.5.4 ♦ 3

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.04	(01)	BI	CA	0565/02	88-10-03	Support for static *universal_integer* expressions

2.4.1 Decimal Literals

A decimal literal is a numeric literal expressed in the conventional decimal notation (that is, the base is implicitly ten).[3] 1

decimal_literal ::= integer [exponent] ♦ 2

integer ::= digit {[underline] digit}

exponent ::= E [+] integer ♦

An underline character inserted between adjacent digits of a decimal literal does not affect the value of this numeric literal. The letter E of the exponent, if any, can be written either in lower case or in upper case, with the same meaning. 3

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. ♦ 4

Examples: 5

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

♦

Notes:

Leading zeros are allowed. No space is allowed in a numeric literal, not even between constituents of the exponent, since a space is a separator. A zero exponent is allowed for an integer literal. 6

References: digit 2.1, lower case letter 2.1, numeric literal 2.4, separator 2.2, space character 2.1, upper case letter 2.1 7

2.4.2 Based Literals

A based literal is a numeric literal expressed in a form that specifies the base explicitly. The base must be at least two and at most sixteen. 1

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

base ::= integer

based_integer ::=
  extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter

An underline character inserted between adjacent digits of a based literal does not affect the value of this numeric literal. The base and the exponent, if any, are in decimal notation. The only letters allowed as extended digits are the letters A through F for the digits ten through fifteen. A letter in a based literal 3

---

[3]Note that the occurrences of *integer* in the productions defining decimal literals might better be changed to *natural* [AI-00052]. See also discussion of ‘‘range -1..10’’ in [AI-00148].

(either an extended digit or the letter E of an exponent) can be written either in lower case or in upper case, with the same meaning.

The conventional meaning of based notation is assumed; in particular the value of each extended digit of a based literal must be less than the base. 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. 4

Examples: 5

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

♦

References: digit 2.1, exponent 2.4.1, letter 2.3, lower case letter 2.1, numeric literal 2.4, upper case letter 2.1 6

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.04.02	(04)	co	WJ	0008/05	86-07-23	Negative exponents in based notation

2.5 Character Literals

A character literal is formed by enclosing one of the 95 graphic characters (including the space) between two apostrophe characters. A character literal has a value that belongs to a character type. 1

character_literal ::= ’graphic_character’ 2

Examples: 3

’A’ ’*’ ’’’ ’ ’

References: character type 3.5.2, graphic character 2.1, literal 4.2, space character 2.1 4

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.05	(01)	ST	RE	0420/03	88-11-08	Allow 256 values for type CHARACTER

2.6 String Literals

A string literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation characters used as *string brackets*. 1

string_literal ::= "{graphic_character}" 2

A string literal has a value that is a sequence of character values corresponding to the graphic characters of the string literal apart from the quotation character itself. If a quotation character value is to be represented in the sequence of character values, then a pair of adjacent quotation characters must be written at the corresponding place within the string literal. (This means that a string literal that includes two adjacent quotation characters is never interpreted as two adjacent string literals.) 3

The *length* of a string literal is the number of character values in the sequence represented. (Each doubled 4

quotation character is counted as a single character.)

Examples:

```
"Message of the day:"
""                          -- an empty string literal
" "    "A"    """"         -- three string literals of length 1

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

Note:

A string literal must fit on one line since it is a lexical element (see 2.2). Longer sequences of graphic character values can be obtained by catenation of string literals. Similarly catenation of constants declared in the package ASCII can be used to obtain sequences of character values that include nongraphic character values (the so-called control characters). Examples of such uses of catenation are given below:

```
"FIRST PART OF A SEQUENCE OF CHARACTERS " &
"THAT CONTINUES ON THE NEXT LINE"

"sequence that includes the" & ASCII.ACK & "control character"
```

References: ascii predefined package C, catenation operation 4.5.3, character value 3.5.2, constant 3.2.1, declaration 3.1, end of a line 2.2, graphic character 2.1, lexical element 2.2

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.06	(06)	ra	WA	0017/05	88-06-17	Use of a string value in a multidimensional aggregate

2.7 Comments

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a program. The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the effect of a program; their sole purpose is the enlightenment of the human reader.

Examples:

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

end; -- processing of LINE is complete

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

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

Note:

Horizontal tabulation can be used in comments, after the double hyphen, and is equivalent to one or more spaces (see 2.2).

References: end of a line 2.2, illegal 1.6, legal 1.6, space character 2.1

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.07	(01)	BI	WJ	0339/04	86-12-01	Allow non-English characters in comments

2.8 Pragmas: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.08	(04)	CR	WJ	0511/05	88-05-23	Error in 0388/04
2.08	(04)	na	na	0185/01	86-02-03	[combined with 0388]
2.08	(04)	ra	WJ	0388/06	88-06-13	Pragmas are allowed in a generic formal part
2.08	(07)	BI	CE	0010/06	88-12-30	The meaning of operations and identifiers in pragma arguments
2.08	(08)	BI	WI	0317/06	87-08-06	Extending language defined pragmas
2.08	(08)	co	WJ	0425/05	87-08-06	Restrictions on arguments of implementation-defined pragmas
2.08	(08)	ra	WI	0011/00	83-10-10	May a pragma affect the m e a n i n g of a legal text outside?
2.08	(09)	BI	CE	0009/04	88-09-02	Implementation-defined names cannot be reserved words
2.08	(09)	BI	CE	0509/01	88-12-08	The legality of pragma arguments
2.08	(09)	BI	WI	0411/01	87-01-20	An object name occurring in a pragma is not a forcing occurrence
2.08	(09)	BI	WJ	0186/08	86-12-04	Pragmas recognized by an impl do not force default representation
2.08	(09)	BI	WJ	0242/09	87-06-18	Subprogram names allowed in pragma INLINE
2.08	(09)	BI	WJ	0306/15	88-05-23	Pragma INTERFACE: allowed names and illegalities
2.08	(09)	BI	WJ	0322/02	86-07-23	Forcing occurrences in unknown pragmas
2.08	(09)	BI	WJ	0371/05	86-07-23	Representation clauses containing forcing occurrences

2.9 Reserved Words

The identifiers listed below are called *reserved words* and are reserved for special significance in the language. For readability of this manual, the reserved words appear in lower case boldface. 1

2

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

A reserved word must not be used as a declared identifier. 3

Notes:

Reserved words differing only in the use of corresponding upper and lower case letters are considered as the same (see 2.3). In some attributes the identifier that appears after the apostrophe is identical to some 4

reserved word.

References: attribute 4.1.4, declaration 3.1, identifier 2.3, lower case letter 2.1, upper case letter 2.1

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.09	(01)	BI	CE	0009/04	88-09-02	Implementation-defined names cannot be reserved words

2.10 Allowable Replacements of Characters

The following replacements are allowed for the vertical bar, sharp, and quotation basic characters:

- A vertical bar character (|) can be replaced by an exclamation mark (!) where used as a delimiter.
- The sharp characters (#) of a based literal can be replaced by colons (:) provided that the replacement is done for both occurrences.
- The quotation characters (") used as string brackets at both ends of a string literal can be replaced by percent characters (%) provided that the enclosed sequence of characters contains no quotation character, and provided that both string brackets are replaced. Any percent character within the sequence of characters must then be doubled and each such doubled percent character is interpreted as a single percent character value.

These replacements do not change the meaning of the program.

Notes:

It is recommended that use of the replacements for the vertical bar, sharp, and quotation characters be restricted to cases where the corresponding graphical symbols are not available. Note that the vertical bar appears as a broken bar on some equipment; replacement is not recommended in this case.

The rules given for identifiers and numeric literals are such that lower case and upper case letters can be used indifferently; these lexical elements can thus be written using only characters of the basic character set. If a string literal of the predefined type STRING contains characters that are not in the basic character set, the same sequence of character values can be obtained by catenating string literals that contain only characters of the basic character set with suitable character constants declared in the predefined package ASCII. Thus the string literal "AB$CD" could be replaced by "AB" & ASCII.DOLLAR & "CD". Similarly, the string literal "ABcd" with lower case letters could be replaced by "AB" & ASCII.LC_C & ASCII.LC_D.

References: ascii predefined package C, based literal 2.4.2, basic character 2.1, catenation operation 4.5.3, character value 3.5.2, delimiter 2.2, graphic character 2.1, graphical symbol 2.1, identifier 2.3, lexical element 2.2, lower case letter 2.1, numeric literal 2.4, string bracket 2.6, string literal 2.6, upper case letter 2.1

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
2.10	(05)	ra	WJ	0350/04	86-12-04	Lexical elements not changed by allowable character replacements

Chapter 3
DECLARATIONS AND TYPES

This chapter 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 entities that are declared, either explicitly or implicitly, by declarations. Such an entity can be a numeric literal, an object, ♦ a record component, a loop parameter, ♦ a type, a subtype, a subprogram, a package, ♦ a formal parameter (of a subprogram ♦), ♦ or an operation (in particular, an attribute or an enumeration literal; see 3.3.3 and 7.4.2). 1

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

basic_declaration ::= 3
    inner_declaration
  | type_declaration | subtype_declaration
  | subprogram_declaration | package_declaration | ♦
  | renaming_declaration | deferred_constant_declaration

inner_declaration ::=
    object_declaration | number_declaration

Certain forms of declaration always occur (explicitly) as part of a basic declaration; these forms are ♦ component declarations ♦ , parameter specifications ♦ , and enumeration literal specifications. A loop parameter specification is a form of declaration that occurs only in certain forms of loop statement. 4

The remaining forms of declaration are implicit ♦. Certain operations are implicitly declared (see 3.3.3). 5

For each form of declaration the language rules define a certain region of text called the *scope* of the declaration (see 8.2). Several forms of declaration 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 associated declared 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 (its simple name); the name is said to *denote* the associated entity and the declaration of the entity is also said to declare the name [AI-00097]. 6

Certain forms of enumeration literal specification associate a character literal with the corresponding declared entity. Certain forms of declaration associate an operator symbol or some other notation with an explicitly or implicitly declared operation. 7

The process by which a declaration achieves its effect is called the *elaboration* of the declaration; this process happens during program execution. 8

After its elaboration, a declaration is said to be *elaborated*. Prior to the completion of its elaboration (including before the elaboration), the declaration is not yet elaborated. The elaboration of any declaration has always at least the effect of achieving this change of state (from not yet elaborated to elaborated). The phrase "*the elaboration has no other effect*" is used in this manual whenever this change of state is the only effect of elaboration for some form of declaration. An elaboration process is also defined for declarative parts, declarative items, and compilation units (see 3.9 and 10.5). 9

Object, number, type, and subtype declarations are described here. The remaining basic declarations are described in later chapters. 10

Note:

The syntax rules use the term *identifier* for the first occurrence of an identifier in some form of declaration; the term *simple name* is used for any occurrence of an identifier that already denotes some declared entity. 11

References: attribute 4.1.4, ♦ block statement 5.6, character literal 2.5, component declaration 3.7, declarative item 3.9, declarative part 3.9, deferred constant declaration 7.4 ♦ , elaboration 3.9 ♦ , enumeration literal specification 3.5.1, ♦ identifier 2.3, ♦ loop parameter specification 5.5, loop statement 5.5, name 4.1, number declaration 3.2.2, numeric literal 2.4, object declaration 3.2.1, operation 3.3, operator symbol 6.1, package declaration 7.1, parameter specification 6.1, record component 3.7, renaming declaration 8.5 , scope 8 .2 , simple name 4.1 , subprogram body 6.3 , subprogram declaration 6 .1 , subtype declaration 3.3.2 , type declaration 3.3.1 , visibility 8.3 12

3.2 Objects and Named Numbers

An *object* is an entity that contains (has) a value of a given type. An object is one of the following: 1

- an object declared by an object declaration ♦, 2
- a formal parameter of a subprogram ♦, 3
- ♦ 4
- a loop parameter, 5
- ♦ 6
- a component ♦ of another object. 7

A number declaration is similar to [AI-00263] an object declaration and is similarly elaborated. It associates an identifier with a value of type *universal_integer* ♦. 8

object_declaration ::= 9
    identifier_list : [**constant**] subtype_indication := expression;
  | ♦

number_declaration ::=
    identifier_list : **constant** := *universal_static*_expression;

identifier_list ::= identifier {, identifier}

An object declaration is called a *single object declaration* if its identifier list has a single identifier; it is called a *multiple object declaration* if the identifier list has two or more identifiers. A multiple object declaration is equivalent to a sequence of the corresponding number of single object declarations. For each identifier of the list, the equivalent sequence has a single object declaration formed by this identifier, followed by a colon and by whatever appears at the right of the colon in the multiple object declaration; the equivalent sequence is in the same order as the identifier list. 10

A similar equivalence applies also for the identifier lists of number declarations, component declarations, parameter specifications, ♦ and deferred constant declarations. 11

In the remainder of this reference manual, explanations are given for declarations with a single identifier; the corresponding explanations for declarations with several identifiers follow from the equivalence stated above. 12

Example: 13

```
-- the multiple object declaration

BUICK, FORD : CAR := (NUMBER => 30300, OWNER => "Smith, Michael K.   "); --  see 3.7

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

BUICK : CAR := (NUMBER => 30300, OWNER => "Smith, Michael K.   ");
FORD  : CAR := (NUMBER => 30300, OWNER => "Smith, Michael K.   ");
```

References: constrained array definition 3.6, component 3.3, declaration 3.1, deferred constant declaration 7.4, ♦ expression 4.4, formal parameter 6.1, identifier 2.3, loop parameter 5.5, numeric type 3.5, parameter specification 6.1, scope 8.2, simple name 4.1, static expression 4.9, subprogram 6, subtype indication 3.3.2, type 3.3, universal_integer type 3.5.4 14

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.02	(08)	BI	WJ	0263/06	88-05-23	A named number is not an object
3.02	(09)	ST	RE	0538/00	87-08-05	Declaring constant arrays with an anonymous type

3.2.1 Object Declarations

An object declaration declares an object whose type is given ♦ by a subtype indication ♦. The object declaration includes the assignment compound delimiter followed by an expression which specifies an initial value for the declared object; the type of the expression must be that of the object. 1

The declared object is a *constant* if the reserved word **constant** appears in the object declaration. The value of a constant cannot be modified after initialization. Formal parameters of mode **in** of subprograms ♦ are also constants; a loop parameter is a constant within the corresponding loop; a subcomponent ♦ of a constant is a constant. 2

An object that is not a constant is called a *variable* (in particular, the object declared by an object declaration that does not include the reserved word **constant** is a variable). The only ways to change the value of a variable are either directly by an assignment, or indirectly when the variable is updated (see 6.2) by a procedure ♦ (this action can be performed either on the variable itself, on a subcomponent of the variable, or on another variable that has the given variable as subcomponent). 3

The elaboration of an object declaration proceeds as follows: 4

(a) The subtype indication ♦ is first elaborated. This establishes the subtype of the object. 5

(b) ♦ The initial value is obtained by evaluating the corresponding expression. ♦ 6

(c) The object is created. 7

(d) The initial value ♦ is assigned to the object ♦. 8

♦ 9

The steps (a) to (d) are performed in the order indicated. ♦ 15

The initialization of an object (the declared object or one of its subcomponents) checks that the initial value belongs to the subtype of the object; for an array object declared by an object declaration, an implicit subtype conversion is first applied as for an assignment statement ♦. The exception CONSTRAINT_ERROR is raised if this check fails. If the object is a constant whose subtype is an unconstrained array type, the initial value must unambiguously determine the array constraints. See 5.2. 16

♦

Examples of variable declarations: 19

```
♦
SIZE            : INTEGER range 0 .. 10_000 := 0;
SORTED          : BOOLEAN := FALSE;
♦
OPTION          : BIT_VECTOR(0..10) := (others => TRUE);
```

Examples of constant declarations: 20

```
LIMIT           : constant INTEGER := 10_000;
LOW_LIMIT       : constant INTEGER := LIMIT/10;
♦
```

Note:

The expression initializing a constant object need not be a static expression (see 4.9). In the above examples, LIMIT and LOW_LIMIT are initialized with static expressions. ♦ 21

References: assignment 5.2, assignment compound delimiter 5.2, component 3.3, composite type 3.3, constrained subtype 3.3, constraint_error exception 11.1, conversion 4.6, declaration 3.1, ♦ elaboration 3.9, evaluation 4.5, expression 4.4, formal parameter 6.1, in some order 1.6, ♦ mode in 6.1, package 7, predefined operator 4.5, primary 4.4, private type 7.4, qualified expression 4.7, reserved word 2.9, scalar type 3.5, subcomponent 3.3, subprogram 6, subtype 3.3, subtype indication 3.3.2, type 3.3, visible part 7.2 22

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.02.01	(01)	BI	CE	0270/04	88-05-09	The type of an object declared to have a private type
3.02.01	(06)	BI	RE	0364/00	85-07-21	subcomponents to components
3.02.01	(16)	BI	WJ	0308/05	88-12-14	Checking default initialization of discriminants for compatibility
3.02.01	(18)	BI	RE	0470/00	86-10-13	Attempt to access undefined component of function result is erroneous
3.02.01	(18)	BI	WJ	0155/08	86-12-01	Evaluation of an attribute prefix having an undefined value
3.02.01	(18)	BI	WJ	0356/08	88-05-23	Access values that designate deallocated objects

3.02.01	(18)	BI	WJ	0374/06	88-05-23	An attempt to access an undefined constant is erroneous
3.02.01	(18)	BI	WJ	0426/05	87-06-18	Operations on undefined array values
3.02.01	(18)	ra	RE	0489/01	88-08-31	Composite assignment with undefined component values
3.02.01	(18)	ra	RE	0490/00	86-11-09	Operations on an undefined variable of a private type.
3.03.01	(03)	ra	CE	0369/06	88-11-20	Representing values of discrete base type

3.2.2 Number Declarations

A number declaration is a special form of constant declaration. The type of the static expression given for the initialization of a number declaration must be the type *universal_integer* ♦. The constant declared by a number declaration is called a *named number* and has the type of the static expression. 1

Note:

The rules concerning expressions of a universal type are explained in section 4.10. It is a consequence of these rules that if every primary contained in the expression is of the type *universal_integer*, then the named number is also of this type. ♦ 2

Examples of number declarations: 3

```
♦
MAX             : constant := 500;      -- a universal integer number
POWER_16        : constant := 2**16;    -- the universal integer 65_536
ONE, UN, EINS   : constant := 1;        -- three different names for 1
MAX_LINE_SIZE   : constant := 120;      -- a universal integer number
```

References: identifier 2.3, primary 4.4, static expression 4.9, type 3.3, universal_integer type 3.5.4, universal type 4.10 4

3.3 Types and Subtypes

A type is characterized by a set of values and a set of operations. 1

There exist several *classes* of types. *Scalar* types are integer types ♦ and types defined by enumeration of their values; values of these types have no components. *Array* and *record* types are composite; a value of a composite type consists of *component* values. ♦ *Private* types are types for which the set of possible values is well defined, but not directly available to the users of such types. ♦ (Private types are described in chapter 7, ♦ the other classes of types are described in this chapter.) 2

♦ 3

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 where the constraint imposes no restriction is also included); a value is said to *satisfy* a constraint if it satisfies the corresponding condition. A *subtype* is a type together with a constraint; a value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself; such a subtype is said to be *unconstrained*: it corresponds to a condition that imposes no restriction. The base type of a type is the type itself. 4

The set of operations defined for a subtype of a given type includes the operations that are defined for the type; however the assignment operation to a variable having a given subtype only assigns values that 5

belong to the subtype. Additional operations, such as qualification (in a qualified expression), are implicitly defined by a subtype declaration.

♦ Certain operations of types and subtypes are called *attributes*; these operations are denoted by the form of name described in section 4.1.4. 6

The term *subcomponent* is used in this manual in place of the term component to indicate either a component, or a component of another component or subcomponent. Where other subcomponents are excluded, the term component is used instead. 7

A value of a given type must not have a subcomponent whose type is the given type itself. 8

The name of a class of types is used in this manual as a qualifier for objects and values that have a type of the class considered. For example, the term "array object" is used for an object whose type is an array type. ♦ 9

Note:

The set of values of a subtype is a subset of the values of the base type. This subset need not be a proper subset; it can be an empty subset. 10

References: array type 3.6, assignment 5.2, attribute 4.1.4, component of an array 3.6, component of a record 3.7, enumeration type 3.5.1, integer type 3.5.4, object 3.2.1, private type 7.4, qualified expression 4.7, record type 3.7, subtype declaration 3.3.2, type declaration 3.3.1 11

3.3.1 Type Declarations

A type declaration declares a type. 1

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

full_type_declaration ::=
    type identifier ♦ is type_definition;

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

The elaboration of a full type declaration consists of ♦ the elaboration of the type definition. 3

The types created by the elaboration of distinct type definitions are distinct types. Moreover, the elaboration of the type definition for a numeric type ♦ creates both a base type and a subtype of the base type; the same holds for a constrained array definition (one of the two forms of array type definition). 4

The simple name declared by a full type declaration denotes the declared type, unless the type declaration declares both a base type and a subtype of the base type, in which case the simple name denotes the subtype, and the base type is anonymous. A type is said to be *anonymous* if it has no simple name. For explanatory purposes, this reference manual sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier. 5

Examples of type definitions: 6

(WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK)
♦
array(1 ..10) **of** INTEGER

Examples of type declarations: 7

type COLOR **is** (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
♦
type TABLE **is array**(0..10) **of** INTEGER;

Notes:

Two type definitions always define two distinct types, even if they are textually identical. ♦ 8

♦ Private type declarations are used in package specifications ♦ (see 7.4 ♦). 10

References: ♦ array type definition 3.6, base type 3.3, constrained array definition 3.6, constrained subtype 3.3, declaration 3.1, elaboration 3.9, enumeration type definition 3.5.1, identifier 2.3, ♦ integer type definition 3.5.4, multiple object declaration 3.2, numeric type 3.5, private type declaration 7.4, reserved word 2.9, type 3.3 11

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.03.01	(03)	ra	CE	0369/06	88-11-20	Representing values of discrete base type

3.3.2 Subtype Declarations

A subtype declaration declares a subtype. 1

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

subtype_indication ::= type_mark [constraint]

type_mark ::= *type*_name | *subtype*_name

constraint ::=
range_constraint | index_constraint | ♦

A type mark denotes a type or a subtype. If a type mark is the name of a type, the type mark denotes this type and also the corresponding unconstrained subtype. The *base type of a type mark* is, by definition, the base type of the type or subtype denoted by the type mark. 3

A subtype indication defines a subtype of the base type of the type mark. 4

If an index constraint appears after a type mark in a subtype indication, the type mark must not already impose an index constraint. ♦ 5

The elaboration of a subtype declaration consists of the elaboration of the subtype indication. The elaboration of a subtype indication creates a subtype. If the subtype indication does not include a constraint, the subtype is the same as that denoted by the type mark. The elaboration of a subtype indication that includes a constraint proceeds as follows: ♦ 6

(a) The constraint is first elaborated. ♦ 7

(b) A check is then made that the constraint is *compatible* with the type or subtype denoted by the type mark. 8

The condition imposed by a constraint is the condition obtained after elaboration of the constraint. (The rules of constraint elaboration are such that the expressions and ranges of constraints are evaluated by the elaboration of these constraints.) The rules defining compatibility are given for each form of constraint in the appropriate section. 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. 9

Examples of subtype declarations: 10

```
subtype RAINBOW is COLOR range RED .. BLUE;          -- see 3.3.1
subtype RED_BLUE  is RAINBOW;
subtype INT is INTEGER;
subtype SMALL_INT is INTEGER range -10 .. 10;
subtype BYTE is BIT_VECTOR(0..7);                    -- see 3.6
♦
```

Note:

A subtype declaration does not define a new type. 11

References: base type 3.3, ♦, compatibility of index constraints 3.6.1, compatibility of range constraints 3.5, constraint_error exception 11.1, declaration 3.1, ♦ elaboration 3.9, evaluation 4.5, expression 4.4, ♦ index constraint 3.6.1, range constraint 3.5, reserved word 2.9, subtype 3.3, type 3.3, type name 3.3.1, unconstrained subtype 3.3 12

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.03.02	(06)	co	RE	0389/00	85-09-16	Creating an already existing subtype
3.03.02	(06)	ra	WJ	0449/04	87-06-18	Evaluating default discriminant expressions

3.3.3 Classification of Operations

The set of operations of a type includes the explicitly declared subprograms that have a parameter or result of the type; such subprograms are necessarily declared after the type declaration or subtype indication.[4] 1

The remaining operations are each implicitly declared for a given type or subtype declaration, immediately after the type or subtype definition. These implicitly declared operations comprise the *basic* operations, the predefined operators (see 4.5), and enumeration literals. ♦ The operations implicitly declared for a given type declaration normally occur immediately after the type declaration and before the next explicit declaration, if any. ♦ 2

A basic operation is an operation that is inherent in one of the following: 3

- An assignment (in assignment statements and initializations), ♦ a membership test, or a short-circuit control form. 4

[4][AI-00624.]

- A selected component or an indexed component ♦. 5
- A qualification (in qualified expressions), an explicit type conversion, or an implicit type conversion of a value of type *universal_integer* to the corresponding value of another numeric type. 6
- A numeric literal (for a universal type), ♦ a string literal, an aggregate, or an attribute. 7

For every type or subtype T, the following attribute is defined: 8

T'BASE — The base type of T. This attribute is allowed only as the prefix of the name of another attribute: for example, T'BASE'FIRST. 9

Notes:

Each literal is an operation whose evaluation yields the corresponding value (see 4.2). Likewise, an aggregate is an operation whose evaluation yields a value of a composite type (see 4.3). Some operations of a type *operate on* values of the type, for example, predefined operators and certain subprograms and attributes. The evaluation of some operations of a type *returns* a value of the type, for example, literals and certain functions, attributes, and predefined operators. Assignment is an operation that operates on an object and a value. The evaluation of the operation corresponding to a selected component ♦ or an indexed component yields the object or value denoted by this form of name. 10

In Ada, *numerous* conversion functions are defined whenever a new integer type or other numeric type is defined. E.g. 10a

```
type T is new parent_type;                                  -- A
  parent_type(x:T) return parent_type;                      -- 1 of these by ARM 3.4(5)
  T(x:parent_type) return T;                                -- 1 of these by ARM 3.4(5)

type T is range L .. R;                                     -- B
  the_one_predefined(x:T) return the_one_predefined;        -- 1 of these by ARM 3.5.4(5),3.4(5)
  T(x:the_one_predefined) return T                          -- 1 of these by ARM 3.5.4(5),3.4(5)
  implicit_T(x:universal_integer) return T;                 -- 1 of these by ARM 3.5.4(8)
```

In B above (type T is range L..R) we don't know *which* predefined integer type was chosen for T to be a subtype of and so we cannot reliably use conversions like SHORT_INTEGER(x). 10b

```
type T is any integer type definition
  T(x:T) return T;                                          -- 1 of these by ARM 4.6(4)

subtype T is integer type specification
  For every conversion of the base type there is one for the subtype that
  converts to the base type and then checks that the result belongs to
  the subtype.                                              -- K of these by ARM 4.6(4).

For all numeric types and subtypes:
  T(x:any_numeric_type) return T;                           -- N of these by ARM 4.6(7)
  any_numeric_type(x:T) return any_numeric_type;            -- N of these by ARM 4.6(7)
```

Note that some of these are operations of existing integer types, e.g. ''integer(x:T) return integer''. While these basic operations of explicit type conversion are visible throughout the entire scope of the associated declaration, the **name** of the type can only be used where it is visible. 10c

References: assignment 5.2, attribute 4.1.4, character literal 2.5, composite type 3.3, conversion 4.6, enumeration literal 3.5.1, formal parameter 6.1, function 6.5, indexed component 4.1.1, initial value 3.2.1, literal 4.2, membership test 4.5 4.5.2, numeric literal 2.4, numeric type 3.5, object 3.2.1, 6.1, predefined operator 4.5, qualified expression 4.7, selected component 4.1.3, short-circuit control form 4.5 4.5.1, string literal 2.6, subprogram 6, subtype 3.3, type 3.3, type declaration 3.3.1, universal_integer type 3.5.4, universal type 4.10 11

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.03.03	(01)	BI	WJ	0330/12	86-07-23	Explicit declaration of enumeration literals
3.03.03	(02)	co	RE	0624/00	88-12-21	Subtype declarations also implicitly declare operations
3.03.03	(07)	BI	RE	0598/00	88-11-08	Of what types is an attribute an operation?

3.4 Derived Types : Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.04	(06)	BI	RE	0567/01	88-09-02	"Corresponding" predefined operation of a derived type
3.04	(10)	BI	WJ	0138/10	87-02-23	Representation clauses for derived types
3.04	(10)	BI	WJ	0292/05	86-12-01	Derived types with address clauses for entries
3.04	(10)	co	RE	0599/00	88-11-08	Representation clauses for derived types
3.04	(11)	BI	RE	0626/00	88-12-21	Within the parenthesis, replace "is" by "must be".
3.04	(11)	BI	WJ	0367/06	88-05-23	Deriving from types declared in a generic package
3.04	(11)	BI	WJ	0398/08	87-06-18	Operations declared for types declared in instances
3.04	(11)	ra	RE	0393/00	85-10-06	Not all operations of a type are derivable
3.04	(13)	BI	WI	0318/03	88-08-17	Conformance rules and derived subprograms
3.04	(14)	co	CE	0433/03	88-11-20	Evaluating and checking parameters "before the call"
3.04	(15)	na	na	0394/03	87-03-11	Is a numeric type a derived type?
3.04	(22)	BI	WJ	0138/10	87-02-23	Representation clauses for derived types

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

range_constraint ::= **range** range 2

range ::= *range*_attribute
| simple_expression .. simple_expression

A range specifies a subset of values of a scalar type. The range L .. R specifies the values from L to R inclusive if the relation L <= R is true. The values L and R are called the *lower bound* and *upper bound* of the range, respectively. A value V is said to *satisfy* a range constraint if it belongs to the range; the value V is said to *belong* to the range if the relations L <= V and V <= R are both TRUE. It is poor programming practice to permit L or R to modify variables that are mutually accessible.[5] A *null* range is a range for which the relation R < L is TRUE; no value belongs to a null range. The operators <= 3

[5]A variable is *accessible* with respect to a particular expression if it is a subexpression of the expression or is accessed by calls on user functions within the expression. A *mutually* accessible variable is one that is accessible to two expressions that may be evaluated in an order not defined by the language.

and < in the above definitions are the predefined operators of the scalar type.

If a range constraint is used in a subtype indication ♦ the type of the simple expressions (likewise, of the bounds of a range attribute) must be the same as the base type of the type mark of the subtype indication. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype, or if the range constraint defines a null range; otherwise the range constraint is not compatible with the subtype. 4

The elaboration of a range constraint consists of the evaluation of the range. The evaluation of a range defines its lower bound and its upper bound. The evaluation of the expressions in L and R proceeds from left to right.[6] 5

Attributes 6

For any scalar type T or for any subtype T of a scalar type, the following attributes are defined: 7

T'FIRST Yields the lower bound of T. The value of this attribute has the same type as T. 8

T'LAST Yields the upper bound of T. The value of this attribute has the same type as T. 9

Note:

Indexing and iteration rules use values of discrete types. 10

References: attribute 4.1.4, constraint 3.3, enumeration type 3.5.1, evaluation 4.5, index 3.6, integer type 3.5.4, loop statement 5.5, range attribute 3.6.2, relational operator 4.5 4.5.2, satisfy a constraint 3.3, simple expression 4.4, subtype indication 3.3.2, type mark 3.3.2 11

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05	(02)	ST	RE	0427/00	86-06-19	Semi-constrained subtypes
3.05	(08)	BI	CA	0174/07	88-10-03	T'FIRST and T'LAST for real types
3.05	(09)	BI	CA	0174/07	88-10-03	T'FIRST and T'LAST for real types

3.5.1 Enumeration Types

An enumeration type definition defines an enumeration type. 1

```
enumeration_type_definition ::=
  (enumeration_literal_specification {, enumeration_literal_specification})

enumeration_literal_specification ::= enumeration_literal

enumeration_literal ::= identifier | character_literal
```
2

The identifiers and character literals listed by an enumeration type definition must be distinct. Each enumeration literal specification is the declaration of the corresponding enumeration literal: this declaration is equivalent to the explicit [AI-00330] declaration of a parameterless function, the designator being the enumeration literal, and the result type being the enumeration type. The designator of such a function is allowed to have the form of a character literal [AI-00401]. The elaboration of an enumeration type definition creates an enumeration type; this elaboration includes that of every 3

[6] **IMPLEMENTATION REQUIREMENT**. The left to right order eliminates one source of incorrect order dependencies.

enumeration literal specification. The implicit body of an enumeration literal is considered to be elaborated when the corresponding enumeration literal specification is elaborated [AI-00430].

Each enumeration literal yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding position numbers. The position number of the value of the first listed enumeration literal is zero; the position number for each other enumeration literal is one more than for its predecessor in the list. 4

If the same identifier or character literal is specified in more than one enumeration type definition, the corresponding 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 must be determinable from the context (see 8.7). 5

Examples: 6

```
type GENDER is (M, F);
type DAY    is (MON, TUE, WED, THU, FRI, SAT, SUN);
type SUIT   is (CLUBS, DIAMONDS, HEARTS, SPADES);
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');
type MIXED  is ('A', 'B', '*', B, NONE, '?', '%');

type MONTH_NAME is (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)

subtype WEEKDAY  is DAY   range MON .. FRI;
subtype MAJOR    is SUIT  range HEARTS .. SPADES;
subtype RAINBOW  is COLOR range RED .. BLUE;  --  the color RED, not the light
```

Note:

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 8.7). 7

References: character literal 2.5, declaration 3.1, designator 6.1, elaboration 3.9, 6.1, function 6.5, identifier 2.3, name 4.1, overloading 6.6 8.7, position number 3.5, qualified expression 4.7, relational operator 4.5 4.5.2, type 3.3, type definition 3.3.1 8

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.01	(03)	BI	RE	0401/00	85-12-03	Character literals are implicitly declared as functions
3.05.01	(03)	BI	WJ	0330/12	86-07-23	Explicit declaration of enumeration literals
3.05.01	(03)	BI	WJ	0430/05	88-05-23	Using an enumeration literal does not raise PROGRAM_ERROR

3.5.2 Character Types

An enumeration type is said to be a character type if at least one of its enumeration literals is a character literal. The predefined type CHARACTER is a character type whose values are the 128 characters of the *ASCII* character set. Each of the 95 graphic characters of this character set is denoted by the corresponding character literal.

Example:

type ROMAN_DIGIT **is** ('I', 'V', 'X', 'L', 'C', 'D', 'M');

Notes:

The predefined package ASCII includes the declaration of constants denoting control characters and of constants denoting graphic characters that are not in the basic character set.

A conventional character set such as *EBCDIC* can be declared as a character type ♦.

References: ascii predefined package C, basic character 2.1, character literal 2.5, constant 3.2.1, declaration 3.1, enumeration type 3.5.1, graphic character 2.1, identifier 2.3, literal 4.2, predefined type C, type 3.3

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.02	(01)	ST	RE	0420/03	88-11-08	Allow 256 values for type CHARACTER

3.5.3 Boolean Types

There is a predefined enumeration type named BOOLEAN. It contains the two literals FALSE and TRUE ordered with the relation FALSE < TRUE.[7] ♦

References: enumeration literal 3.5.1, enumeration type 3.5.1, relational operator 4.5 4.5.2, type 3.3

3.5.4 Integer Types

♦

The predefined integer types include the type INTEGER. ♦

Integer literals are the literals of an anonymous predefined integer type that is called *universal_integer* in this reference manual. Other integer types have no literals. However, for each integer type there exists an implicit conversion that converts a *universal_integer* value into the corresponding value (if any) of the integer type. The circumstances under which these implicit conversions are invoked are described in section 4.6.

The position number of an integer value is the corresponding value of the type *universal_integer*.

The same arithmetic operators are predefined for all integer types (see 4.5). The exception

[7]Note that **type** foo = (FALSE,TRUE) is not a boolean type. It is a distinct enumeration type that overloads FALSE and TRUE.

CONSTRAINT_ERROR[8] is raised by the execution of an operation (in particular an implicit conversion) that cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the integer type). ♦

Examples: 11

```
subtype PAGE_NUM      is INTEGER range 1 .. 2_000;
subtype LINE_SIZE     is INTEGER range 1 .. MAX_LINE_SIZE;

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;
```

Notes:

♦ The predefined operators of an integer type deliver results whose range is defined by the parent predefined type; such a result need not belong to the declared subtype, in which case an attempt to assign the result to a variable of the integer subtype raises the exception CONSTRAINT_ERROR. 12

The smallest (most negative) value supported by the predefined integer types of an implementation (excluding *universal_integer* [AI-000565]) is the named number AVA.MIN_INT and the largest (most positive) value is AVA.MAX_INT (see 13.7). 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.[9] 13

References: anonymous type 3.3.1, belong to a subtype 3.3, bound of a range 3.5, constraint_error exception 11.1, conversion 4.6, identifier 2.3, integer literal 2.4, literal 4.2, numeric_error exception 11.1, parent type 3.4, predefined operator 4.5, range constraint 3.5, static expression 4.9, subtype declaration 3.3.2, system predefined package 13.7, type 3.3, type declaration 3.3.1, type definition 3.3.1, universal type 4.10 14

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.04	(00)	ST	RE	0600/00	88-11-08	Why We Need Unsigned Integers in Ada
3.05.04	(03)	ra	WJ	0240/05	86-07-23	Integer type definitions cannot contain a RANGE attribute
3.05.04	(04)	BI	WJ	0023/06	86-07-23	Static numeric subtypes
3.05.04	(05)	na	na	0394/03	87-03-11	Is a numeric type a derived type?
3.05.04	(07)	BI	CE	0459/04	88-12-07	Precision and range of predefined numeric types
3.05.04	(07)	BI	WI	0597/01	88-11-08	Unsigned integer types can be provided
3.05.04	(07)	ra	CE	0402/05	88-11-20	Unsigned integer types are not predefined
3.05.04	(08)	BI	CA	0565/02	88-10-03	Support for static *universal_integer* expressions
3.05.04	(10)	NB	WJ	0387/05	87-02-23	Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR
3.05.04	(10)	ra	WJ	0267/06	88-07-07	Evaluating expressions in case statements
3.05.04	(13)	BI	CA	0565/02	88-10-03	Support for static *universal_integer* expressions
3.05.04	(13)	BI	WA	0304/05	88-11-04	The definition of SYSTEM.MIN_INT and SYSTEM.MAX_INT

---

[8] AI-00387.

[9] **IMPLEMENTATION REQUIREMENT**. Ada requires that such expressions *be accepted*, unless insufficient resources (memory) are available [AI-00565]. We require otherwise in order that:

(a) we have a single, predictable mathematics

(b) we can reliably determine what is and is not a static expression.

See also 4.10(4).

3.5.5 Operations of Discrete Types

The basic operations of a discrete type include the operations involved in assignment, the membership tests, and qualification; for a boolean type they include the short-circuit control forms; for an integer type they include the explicit conversion of values of other numeric types to the integer type, and the implicit conversion of values of the type *universal_integer* to the type. 1

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. 1a

Finally, for every discrete type or subtype T, the basic operations include the attributes listed below. In this presentation, T is referred to as being a subtype (the subtype T) for any property that depends on constraints imposed by T; other properties are stated in terms of the base type of T. 2

The first group of attributes yield characteristics of the subtype T. This group includes the attribute BASE (see 3.3.3) and the attributes FIRST and LAST (see 3.5) ♦. 3

♦ 4

All attributes of the second group are functions with a single parameter. The corresponding actual parameter is indicated below by X. 5

T'POS — This attribute is a function. The parameter X must be a value of the base type of T. The result type is the type *universal_integer*. The result is the position number of the value of the parameter. 6

T'VAL — This attribute is a ♦ function with a single parameter of base type integer.[10] The result type is the base type of T. The result is the value whose position number is the *universal_integer* value corresponding to X. The exception CONSTRAINT_ERROR is raised if the *universal_integer* value corresponding to X is not in the range T'POS(T'BASE'FIRST) .. T'POS(T'BASE'LAST). 7

T'SUCC — This attribute is a function. The parameter X must be a value of the base type of T. The result type is the base type of T. The result is the value whose position number is one greater than that of X. The exception CONSTRAINT_ERROR is raised if X equals T'BASE'LAST. 8

T'PRED — This attribute is a function. The parameter X must be a value of the base type of T. The result type is the base type of T. The result is the value whose position number is one less than that of X. The exception CONSTRAINT_ERROR is raised if X equals T'BASE'FIRST. 9

T'IMAGE — This attribute is a function. The parameter X must be a value of the base type of T. The result type is the predefined type STRING. The result is the image of the value of X, that is, a sequence of characters representing the value in display form. 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. The lower bound of the index of the image is one. 10

---

[10] We have eliminated the notion of *special function* as it is not needed in our subset. See AI-00013 for a discussion. Note that if the argument to T'VAL is *universal_integer* it gets implicitly converted to integer. This may result in CONSTRAINT_ERROR, but this is the *identical* behavior to the special function.

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. The image of a character C, other than a graphic character, is implementation-defined; the only requirement is that the image must be such that C equals CHARACTER'VALUE(CHARACTER'IMAGE(C)). 11

T'VALUE This attribute is a function. The parameter X must be a value of the predefined type STRING. The result type is the base type of T. Any leading and any trailing spaces of the sequence of characters that corresponds to the parameter are ignored. 12

For an enumeration type, if the sequence of characters has the syntax of an enumeration literal and if this literal exists for the base type of T, the result is the corresponding enumeration value. For the type CHARACTER, if the sequence of characters is the image of a character other than a graphic character, the result is the corresponding enumeration value [AI-00239]. For an integer type, if the sequence of characters has the syntax of an integer literal, with an optional single leading character that is a plus or minus sign, and if there is a corresponding value in the base type of T, the result is this value. In any other case, the exception CONSTRAINT_ERROR is raised. 13

♦ 14

Besides the basic operations, the operations of a discrete type include the predefined relational operators. For enumeration types, operations include enumeration literals. For boolean types, operations include the predefined unary logical negation operator **not**, and the predefined logical operators. For integer types, operations include the predefined *arithmetic* operators: these are the binary and unary adding operators - and +, all multiplying operators, the unary operator **abs**, and the exponentiating operator. 15

The operations of a subtype are the corresponding operations of its base type. Some of the base type operations depend on subtype information in order to execute. These operations are: assignment, membership tests, qualification, explicit type conversions, and the attributes of the first group; the effect of each of these operations depends on the subtype (assignments, membership tests, qualifications, and conversions involve a subtype check; attributes of the first group yield a characteristic of the subtype). 16

Notes:

For a subtype of a discrete type, the results delivered by the attributes SUCC, PRED, VAL, and VALUE need not belong to the subtype; similarly, the actual parameters of the attributes POS, SUCC, PRED, and IMAGE need not belong to the subtype. The following relations are satisfied (in the absence of an exception) by these attributes: 17

```
T'POS(T'SUCC(X)) = T'POS(X) + 1
T'POS(T'PRED(X)) = T'POS(X) - 1

T'VAL(T'POS(X))  = X
T'POS(T'VAL(N))  = N
```

Examples: 18

```
-- For the types and subtypes declared in section 3.5.1 we have:

-- COLOR'FIRST      = WHITE,    COLOR'LAST      = BLACK
-- RAINBOW'FIRST    = RED,      RAINBOW'LAST    = BLUE

-- COLOR'SUCC(BLUE) = RAINBOW'SUCC(BLUE) = BROWN
```

```
-- COLOR'POS(BLUE)     = RAINBOW'POS(BLUE)     = 4
-- COLOR'VAL(0)        = RAINBOW'VAL(0)        = WHITE
```

References: abs operator 4.5 4.5.6, assignment 5.2, attribute 4.1.4, base type 3.3, basic operation 3.3.3, 19
binary adding operator 4.5 4.5.3, boolean type 3.5.3, bound of a range 3.5, character literal 2.5, constraint 3.3, constraint_error exception 11.1, conversion 4.6, discrete type 3.5, enumeration literal 3.5.1, exponentiating operator 4.5 4.5.6, function 6.5, graphic character 2.1, identifier 2.3, integer type 3.5.4, logical operator 4.5 4.5.1, membership test 4.5 4.5.2, multiplying operator 4.5 4.5.5, not operator 4.5 4.5.6, numeric literal 2.4, numeric type 3.5, object 3.2, operation 3.3, position number 3.5, predefined operator 4.5, predefined type C, qualified expression 4.7, relational operator 4.5 4.5.2, short-circuit control form 4.5 4.5.1, string type 3.6.3, subtype 3.3, type 3.3, unary adding operator 4.5 4.5.4, universal_integer type 3.5.4, universal type 4.10

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.05	(07)	BI	WI	0013/01	83-11-08	Is 'VAL renameable?
3.05.05	(10)	BI	WJ	0234/05	87-03-16	Lower bound for 'IMAGE of enumeration values
3.05.05	(11)	NB	WJ	0239/11	87-02-23	ENUMERATION_IO and IMAGE for non-graphic characters

3.5.6 Real Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.06	(06)	NB	WJ	0387/05	87-02-23	Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR

3.5.7 Floating Point Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.07	(00)	co	RE	0457/00	86-09-05	Real type definitions cannot contain a RANGE attribute
3.05.07	(06)	ra	WJ	0205/06	87-06-18	The formula for *mantissa* is correct
3.05.07	(08)	BI	CE	0459/04	88-12-07	Precision and range of predefined numeric types
3.05.07	(09)	co	WJ	0217/05	88-05-23	The safe numbers of a floating point subtype
3.05.07	(09)	co	WJ	0314/05	87-08-06	The safe numbers for IBM-370 floating point
3.05.07	(10)	BI	WJ	0023/06	86-07-23	Static numeric subtypes
3.05.07	(11)	na	na	0394/03	87-03-11	Is a numeric type a derived type?
3.05.07	(12)	ra	CA	0469/03	88-05-10	When the bounds in a type declaration are not model numbers
3.05.07	(15)	BI	WI	0344/05	88-06-30	Model and safe numbers for range-constrained real subtypes
3.05.07	(17)	ra	WJ	0375/05	88-05-23	Restricting the allowed values of a floating point subtype

3.5.8 Operations of Floating Point Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.08	(16)	BI	WJ	0407/06	88-05-23	The operations of a subtype with reduced accuracy

3.5.9 Fixed Point Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.09	(02)	ST	RE	0518/00	87-02-10	Fixed and Floating type Declarations needlessly Different
3.05.09	(05)	BI	RE	0640/00	88-12-21	Small overlooked
3.05.09	(05)	ST	RE	0519/00	87-02-10	"Small" should be a power of two TIMES THE RANGE
3.05.09	(06)	ra	WJ	0143/04	86-07-23	Model numbers for delta 1.0 range -7.0 .. 8.0
3.05.09	(06)	ra	WP	0340/05	88-12-07	Model numbers for fixed point types having a null range
3.05.09	(07)	ra	RE	0520/01	87-02-10	What are Fixed Point BASE types and Predefined types?
3.05.09	(08)	BI	WJ	0023/06	86-07-23	Static numeric subtypes
3.05.09	(09)	BI	CE	0341/13	88-12-07	Extra precision or range for fixed point representations
3.05.09	(09)	BI	WJ	0144/10	87-09-12	A fixed point type declaration cannot raise an exception
3.05.09	(09)	CR	WJ	0471/04	87-09-12	Correction to 0144/08 examples
3.05.09	(09)	na	na	0394/03	87-03-11	Is a numeric type a derived type?
3.05.09	(10)	ra	WJ	0343/05	86-12-01	Decimal fixed point representations
3.05.09	(11)	BI	WJ	0508/03	88-05-23	The safe numbers of a fixed point subtype
3.05.09	(11)	na	na	0428/01	86-08-08	[combined with 0217]
3.05.09	(14)	BI	WI	0344/05	88-06-30	Model and safe numbers for range-constrained real subtypes
3.05.09	(14)	ra	WJ	0145/04	86-07-23	Dynamic computation of 'MANTISSA for fixed point subtypes
3.05.09	(14)	ra	WJ	0146/10	88-05-23	Model numbers for a fixed point subtype with length clause
3.05.09	(14)	ST	RE	0521/00	87-02-10	Fixed Point Subtypes inheriting Small
3.05.09	(16)	ra	WJ	0146/10	88-05-23	Model numbers for a fixed point subtype with length clause
3.05.09	(18)	ra	WJ	0147/05	87-08-06	Declaring a fixed point type that occupies one word

3.5.10 Operations of Fixed Point Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.05.10	(08)	CR	CE	0583/02	88-11-20	Correction to 0179/08
3.05.10	(08)	CR	WJ	0467/04	88-05-23	Correction to 0179/06
3.05.10	(08)	ra	WJ	0179/08	88-06-13	The definition of the attribute FORE
3.05.10	(15)	BI	WJ	0407/06	88-05-23	The operations of a subtype with reduced accuracy

3.6 Array Types

An array object is a composite object consisting of components that have the same subtype. The name for a component of an array uses one or more index values belonging to specified integer[11] types. The value of an array object is a composite value consisting of the values of its components. 1

array_type_definition ::= 2
  unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
  array(index_subtype_definition {, index_subtype_definition}) **of**
    *component*_subtype_indication

constrained_array_definition ::=
  array index_constraint **of** *component*_subtype_indication

---

[11]"Discrete" type was changed to integer type so that we don't need to worry about constraint errors arising from null strings with indices of enumerated types without predecessors for the lower bound. This did not quite work, since we still need to check for null strings of types like STR:

type STR **is array**(INTEGER **range** <>) **of** CHARACTER;

See also 4.2.

index_subtype_definition ::= *index*_type_mark **range** <>

index_constraint ::= (index_range {, index_range})

index_range ::= *integer*_subtype_indication | *integer*_range

discrete_range ::= *discrete*_subtype_indication | range

An array object 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 type and possible constraint of the components. The order of the indices is significant. 3

A one-dimensional array has a distinct component for each possible index value. A multidimensional array has a distinct component for each possible sequence of index values that can be formed by selecting one value for each index position (in the given order). The possible values for a given index are all the values between the lower and upper bounds, inclusive; this range of values is called the *index range*. 4

An unconstrained array definition defines an array type. For each object that has the 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, except for null arrays as explained in section 3.6.1. The *index subtype* for a given index position is, by definition, the subtype denoted by the type mark of the corresponding index subtype definition. 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). The elaboration of an unconstrained array definition creates an array type; this elaboration includes that of the component subtype indication. 5

A constrained array definition defines both an array type and a subtype of this type: 6

- The array type is an implicitly declared anonymous type; this type is defined by an (implicit) unconstrained array definition, in which the component subtype indication is that of the constrained array definition, and in which the type mark of each index subtype definition denotes the subtype defined by the corresponding discrete range. 7
- The array subtype is the subtype obtained by imposition of the index constraint on the array type. 8

If a constrained array definition is given for a type declaration, the simple name declared by this declaration denotes the array subtype. 9

The elaboration of a constrained array definition creates the corresponding array type and array subtype. For this elaboration, the index constraint and the component subtype indication are elaborated. The elaboration of each discrete range of the index constraint proceeds left to right followed by the elaboration of the component subtype indication.[12] It is poor programming practice to permit index constraints and/or the component subtype indication to modify variables that are mutually accessible. 10

Examples of type declarations with unconstrained array definitions: 11

♦
type BIT_VECTOR **is array**(INTEGER **range** <>) **of** BOOLEAN;
type MATRIX **is array**(INTEGER **range** <>,INTEGER **range** <>) **of** INTEGER;

---

[12]**IMPLEMENTATION REQUIREMENT**. The required order eliminates one source of incorrect order dependencies.

```
type ROMAN        is array(POSITIVE range <>) of ROMAN_DIGIT;
```

Examples of type declarations with constrained array definitions: 12

```
type SCHEDULE is array(1..7) of BOOLEAN;
type TABLE    is array(1 .. 10) of INTEGER;
type LINE     is array(1 .. MAX_LINE_SIZE) of CHARACTER;
```

Examples of object declarations with constrained array definitions: 13

```
GRID :  array(1 .. 80, 1 .. 100) of BOOLEAN;
MIX  :  array(COLOR range RED .. GREEN) of BOOLEAN;
PAGE : array(1..50) of LINE;  --  an array of arrays
```

Note:

For a one-dimensional array, the rule given means that a type declaration with a constrained array definition such as 14

```
type T is array (POSITIVE range  MIN .. MAX) of COMPONENT;
```

is equivalent (when legal) to the succession of declarations 15

```
subtype index_subtype is POSITIVE range MIN .. MAX;
type array_type is array(INDEX_SUBTYPE range <>) of COMPONENT;
subtype T is array_type(index_subtype);
```

where *index_subtype* and *array_type* are both anonymous. Consequently, T is the name of a subtype and all objects declared with this type mark are arrays that have the same bounds. Similar transformations apply to multidimensional arrays. 16

♦ 17

References: anonymous type 3.3.1, bound of a range 3.5, component 3.3, constraint 3.3, discrete type 3.5, elaboration 3.1 3.9, in some order 1.6, name 4.1, object 3.2, range 3.5, subtype 3.3, subtype indication 3.3.2, type 3.3, type declaration 3.3.1, type definition 3.3.1, type mark 3.3.2 18

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.06	(01)	co	RE	0417/00	86-04-15	Allowed range of index subtypes
3.06	(05)	BI	RE	0249/00	84-05-26	Index types should be required to be discrete.
3.06	(07)	ra	CE	0369/06	88-11-20	Representing values of discrete base type

3.6.1 Index Constraints and Discrete Ranges

An index constraint determines the range of possible values for every index of an array type, and thereby the corresponding array bounds. 1

For an integer range used in a constrained array definition and defined by a range, an implicit conversion to the predefined type INTEGER is assumed if each bound is either a numeric literal, a named number, or an attribute, and the type of both bounds (prior to the implicit conversion) is the type *universal_integer*. Otherwise, both bounds must be of the same integer type (i.e., INTEGER, by 3.6), other than *universal_integer*; this type must be determinable independently of the context, but using the fact that the 2

type must be discrete and that both bounds must have the same type. These rules apply also to a discrete range used in an iteration rule (see 5.5) ♦.[13]

If an index constraint follows a type mark in a subtype indication, then the type or subtype denoted by the type mark must not already impose an index constraint. The type mark must denote an unconstrained array type ♦. The index constraint must provide an integer range for each index of the array type and the type of each integer range must be the same as that of the corresponding index. 3

An index constraint is *compatible* with the type denoted by the type mark if and only if the constraint defined by each discrete range is compatible 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. (Note, however, that assignment and certain other operations on arrays involve an implicit subtype conversion.) 4

The bounds of each array object are determined as follows: 5

- For a variable declared by an object declaration, the subtype indication of the corresponding object declaration must define a constrained array subtype (and, thereby, the bounds). The same requirement exists for the subtype indication of a component declaration, if the type of the record component is an array type; and for the component subtype indication of an array type definition, if the type of the array components is itself an array type. 6
- For a constant declared by an object declaration, the bounds of the constant are defined by the initial value if the subtype of the constant is unconstrained; they are otherwise defined by this subtype (in the latter case, the initial value is the result of an implicit subtype conversion). 7
- ♦ 8
- For a formal parameter of a subprogram ♦ the bounds are obtained from the corresponding actual parameter. (The formal parameter is constrained with the corresponding values of the bounds.) 9
- For a renaming declaration ♦ the bounds are those of the renamed object ♦. 10

For the elaboration of an index constraint, the discrete ranges are evaluated left to right.[14] It is poor programming practice to permit the different index ranges to modify variables that are mutually accessible. 11

Examples of array declarations including an index constraint: 12

```
RECTANGLE : MATRIX(1 .. 20, 1 .. 30);
BOARD     : MATRIX(1 .. 8,  1 .. 8);  --  see 3.6
INVERSE   : MATRIX(1 .. N,  1 .. N);  --  N need not be static

FILTER    : BIT_VECTOR(0 .. 31);
```

♦ 13

Notes:

---

[13]The range ‘‘-1..10’’ is illegal as an integer range in a constrained array definition (as well as in an iteration rule) since its elements are not both ‘‘numeric literals, named numbers, or attributes’’ of type *universal_integer*. The independently determined types of these expressions are *universal_integer*. Therefore, the range is illegal. Basically, this is due to the fact that -1 is not a *universal_integer* literal [AI-00148].

[14]*IMPLEMENTATION REQUIREMENT*. The left to right order eliminates one source of incorrect order dependencies.

The elaboration of a subtype indication consisting of a type mark followed by an index constraint checks the compatibility of the index constraint with the type mark (see 3.3.2). 15

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. 16

References: actual parameter 6.4.1, array bound 3.6, array component 3.6, array type 3.6, array type definition 3.6, bound of a range 3.5, compatible 3.3.2, component declaration 3.7, constant 3.2.1, constrained array definition 3.6, constrained array subtype 3.6, conversion 4.6, ♦ discrete range 3.6, expression 4.4, formal parameter 6.1, function 6.5, index 3.6, index constraint 3.6.1, index subtype 3.6, initial value 3.2.1, integer literal 2.4, integer type 3.5.4, iteration rule 5.5, mode 12.1.1, name 4.1, null range 3.5, object 3.2, object declaration 3.2.1, predefined type C, range 3.5, record component 3.7, renaming declaration 8.5, result subtype 6.1, satisfy 3.3, subprogram 6, subtype conversion 4.6, subtype indication 3.3.2, type mark 3.3.2, unconstrained array type 3.6, unconstrained subtype 3.3, universal type 4.10, universal_integer type 3.5.4, variable 3.2.1 17

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.06.01	(02)	BI	WA	0218/08	88-11-08	Implicit conversion of attributes that are functions
3.06.01	(02)	ra	RE	0161/01	84-01-13	Index Constraints with mixed bounds
3.06.01	(02)	ra	WJ	0148/05	86-07-23	Legality of -1..10 in loops
3.06.01	(02)	ST	RE	0140/01	85-09-16	Allow -1..10 as a discrete range in loops
3.06.01	(04)	co	WJ	0282/06	86-12-01	Compatibility of constraint defined by discrete range

3.6.2 Operations of Array Types

The basic operations[15] of an array type include the operations involved in assignment and aggregates ♦, membership tests, indexed components, qualification, and explicit conversion; for one-dimensional arrays the basic operations also include the operations involved in ♦ string literals if the component type is a character type. 1

If A is an array object, an array value, or a constrained array subtype, the basic operations also include the attributes listed below. These attributes are not allowed for an unconstrained array type. The argument N used in the attribute designators for the N-th dimension of an array must be a static expression of type *universal_integer*. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. 2

A’FIRST — Yields the lower bound of the first index range. The value of this attribute has the same type as this lower bound. 3

A’FIRST(N) — Yields the lower bound of the N-th index range. The value of this attribute has the same type as this lower bound. 4

A’LAST — Yields the upper bound of the first index range. The value of this attribute has the same type as this upper bound. 5

A’LAST(N) — Yields the upper bound of the N-th index range. The value of this attribute has the same type as this upper bound. 6

A’RANGE — Yields the first index range, that is, the range A’FIRST .. A’LAST. 7

---

[15]Some of operations of an array type require or return information about the constraints of the subtype. In this case we talk about operations or attributes *of the subytype*, but formally these operations are operations of the *base* type that take additional arguments carrying the needed constraint information. See also 3.5.5.

A’RANGE(N)	Yields the N-th index range, that is, the range A’FIRST(N) .. A’LAST(N).	8
A’LENGTH	Yields the number of values of the first index range (zero for a null range). The value of this attribute is of the type *universal_integer*.	9
A’LENGTH(N)	Yields the number of values of the N-th index range (zero for a null range). The value of this attribute is of the type *universal_integer*.	10

In addition, the attribute T’BASE is defined for an array type or subtype T (see 3.3.3). ♦ 11

Besides the basic operations, the operations of an array type include the predefined comparison for equality and inequality♦. For one-dimensional arrays, the operations include catenation♦. ♦ 12

Examples (using arrays declared in the examples of section 3.6.1): 13

```
-- FILTER’FIRST            =  0   FILTER’LAST            = 31   FILTER’LENGTH    = 32
-- RECTANGLE’LAST(1)  =  20   RECTANGLE’LAST(2)  = 30
```

Notes:

The attributes A’FIRST and A’FIRST(1) yield the same value. A similar relation exists for the attributes A’LAST, A’RANGE, and A’LENGTH. The following relations are satisfied (except for a null array) by the above attributes if the index type is an integer type: 14

```
A’LENGTH      = A’LAST    - A’FIRST    + 1
A’LENGTH(N)   = A’LAST(N) - A’FIRST(N) + 1
```

♦ 15

References: aggregate 4.3, array type 3.6, assignment 5.2, attribute 4.1.4, basic operation 3.3.3, bound of a range 3.5, catenation operator 4.5 4.5.3, character type 3.5.2, constrained array subtype 3.6, conversion 4.6, designator 6.1, dimension 3.6, index 3.6, indexed component 4.1.1, ♦ membership test 4.5 4.5.2, null range 3.5, object 3.2, operation 3.3, predefined operator 4.5, qualified expression 4.7, relational operator 4.5 4.5.2, static expression 4.9, string literal 2.6, subcomponent 3.3, type 3.3, unconstrained array type 3.6, universal type 4.10, universal_integer type 3.5.4 16

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.06.02	(02)	co	WA	0271/04	88-06-16	Using array attributes with access values
3.06.02	(02)	ST	RE	0584/00	88-09-02	Restrict argument of RANGE attribute in Ada 9x

3.6.3 The Type String

The values of the predefined type STRING are one-dimensional arrays of the predefined type CHARACTER, indexed by values of the predefined subtype POSITIVE: 1

```
subtype POSITIVE is INTEGER range 1 .. INTEGER’LAST;
type STRING is array(POSITIVE range <>) of CHARACTER;
```

Examples: 2

```
STARS      : STRING(1 .. 120) := (1 .. 120 => ’*’ );
QUESTION   : constant STRING  := "HOW MANY CHARACTERS?";
--  QUESTION’FIRST = 1, QUESTION’LAST = 20 (the number of characters)
```

```
ASK_TWICE     : constant STRING  := QUESTION & QUESTION;
NINETY_SIX    : constant ROMAN   := "XCVI";       -- see 3.6
```

Notes:

String literals (see 2.6 and 4.2) are basic operations applicable to the type STRING and to any other one-dimensional array type whose component type is a character type. The catenation operator is a predefined operator for the type STRING and for one-dimensional array types; it is represented as &. The relational operators <, <=, >, and >= are defined for values of these types, and correspond to lexicographic order (see 4.5.2). 3

References: aggregate 4.3, array 3.6, catenation operator 4.5 4.5.3, character type 3.5.2, component type (of an array) 3.6, dimension 3.6, index 3.6, lexicographic order 4.5.2, positional aggregate 4.3, predefined operator 4.5, predefined type C, relational operator 4.5 4.5.2, string literal 2.6, subtype 3.3, type 3.3 4

3.7 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 its components. 1

```
record_type_definition ::=                                          2
  record
    component_list
  end record

component_list ::=
    component_declaration {component_declaration}
  | ♦

component_declaration ::=
  identifier_list : component_subtype_definition ♦ ;

component_subtype_definition ::=  subtype_indication
```

Each component declaration declares a component of the record type. ♦ The identifiers of all components of a record type must be distinct. The use of a name that denotes a record component ♦ is not allowed within the record type definition that declares the component. 3

A component declaration with several identifiers is equivalent to a sequence of single component declarations, as explained in section 3.2. Each single component declaration declares a record component whose subtype is specified by the component subtype definition. 4

♦ 5

♦ The same components are present in all values of the record type. ♦ 7

The elaboration of a record type definition creates a record type; it consists of the elaboration of any corresponding (single) component declarations, in the order in which they appear ♦. The elaboration of a component declaration consists of the elaboration of the component subtype definition. 8

For the elaboration of a component subtype definition, ♦ the subtype indication is elaborated. ♦ 9

Examples of record type declarations: 10

```
type DATE is
  record
    DAY        : INTEGER range 1 .. 31;
    MONTH      : MONTH_NAME;
    YEAR       : INTEGER range 0 .. 4000;
  end record;

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: 11

```
TOMORROW, YESTERDAY : DATE := (1, JANUARY, 1988);
NEXT_CAR : CAR := (34549821,"Smith, Michael K.   ");
NEXT_PERSON : PERSON := ("Smith, Michael K.   ",YESTERDAY,40,
                                    NEXT_CAR,"Smith, Elizabeth B. ")
```

Notes:

♦ 12

Unlike the components of an array, the components of a record need not be of the same type. 12

References: assignment compound delimiter 2.2, component 3.3, composite value 3.3, ♦ declaration 3.1, elaboration 3.9, expression 4.4, identifier 2.3, identifier list 3.2, ♦ name 4.1, object 3.2, subtype 3.3, type 3.3, type mark 3.3.2 13

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.07	(00)	ra	CE	0566/05	88-10-03	Storage allocation for constrained record subtypes
3.07	(02)	ST	RE	0429/00	86-06-19	Allow array type definition for record component
3.07	(08)	BI	WJ	0358/10	86-12-04	Discriminant checks for non-existent subcomponents

3.7.1 Discriminants: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.07.01	(03)	BI	RE	0654/00	88-12-21	Derived type overlooked
3.07.01	(06)	ra	WI	0175/01	86-01-28	Discriminant name as part of larger expression

3.7.2 Discriminant Constraints: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.07.02	(01)	ra	RE	0162/00	84-01-13	Discriminant Constraints do not apply to subtypes
3.07.02	(05)	BI	WJ	0007/19	86-12-04	Discriminant checks for incomplete, private, and access types
3.07.02	(05)	BI	WJ	0358/10	86-12-04	Discriminant checks for non-existent subcomponents
3.07.02	(05)	co	WJ	0319/09	87-06-18	Checking for subtype incompatibility
3.07.02	(08)	BI	WJ	0014/10	87-06-18	Evaluating default discriminant expressions
3.07.02	(08)	BI	WJ	0308/05	88-12-14	Checking default initialization of discriminants for compatibility
3.07.02	(08)	co	RE	0456/00	86-08-12	Required discriminant constraint
3.07.02	(10)	BI	WJ	0308/05	88-12-14	Checking default initialization of discriminants for compatibility

3.7.3 Variant Parts: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.07.03	(00)	ST	RE	0345/00	85-06-18	Record type with variant having no discriminants

3.7.4 Operations of Record Types

The basic operations of a record type include the operations involved in assignment and aggregates ♦, membership tests, selection of record components, and qualification. 1

In addition, the attribute T'BASE ♦ is defined for a record type T (see 3.3.3). 4

Besides the basic operations, the operations of a record type include the predefined comparison for equality and inequality♦. 5

♦ 6

References: actual parameter 6.4.1, aggregate 4.3, assignment 5.2, attribute 4.1.4, basic operation 3.3.3, boolean type 3.5.3, constant 3.2.1, conversion 4.6, formal parameter 6.1, ♦ membership test 4.5 4.5.2, mode 6.1, object 3.2.1, operation 3.3, predefined operator 4.5, predefined type C, qualified expression 4.7, record type 3.7, relational operator 4.5 4.5.2, selected component 4.1.3, subcomponent 3.3, subtype 3.3, type 3.3 7

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.07.04	(03)	ra	WA	0005/08	88-11-04	'CONSTRAINED for a formal parameter

3.8 Access Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.08	(06)	BI	WJ	0324/08	88-05-23	Checking the subtype of a non-null access value
3.08.01	(04)	BI	WJ	0007/19	86-12-04	Discriminant checks for incomplete, private, and access types
3.08.01	(04)	co	WJ	0319/09	87-06-18	Checking for subtype incompatibility
3.08.01	(04)	na	na	0264/01	84-11-05	[combined with 0039]
3.08.01	(04)	ra	WJ	0231/05	88-05-23	Full declarations of incomplete types can have discriminants
3.08.02	(02, 03)	BI	RE	0101/00	83-11-07	A prefix cannot be a value.

3.08.02 (04) na na 0275/08 86-04-16 [Combined with 0007]

3.9 Declarative Parts

A declarative part contains declarative items (possibly none). 1

declarative_part ::= {basic_declarative_item} {later_declarative_item} 2

basic_declarative_item ::= basic_declaration | ♦ | use_clause

later_declarative_item ::= subprogram_body | package_body
| subprogram_declaration | package_declaration |♦
| use_clause | ♦

inner_declarative_part ::= {inner_declarative_item}

inner_declarative_item ::= inner_declaration | use_clause

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. After its elaboration, a declarative item is said to be *elaborated*. Prior to the completion of its elaboration (including before the elaboration), the declarative item is not yet elaborated. 3

For several forms of declarative item, the language rules (in particular scope and visibility rules) are such that it is either impossible or illegal to use an entity before the elaboration of the declarative item that declares this entity. For example, it is not possible to use the name of a type for an object declaration if the corresponding type declaration is not yet elaborated. In the case of bodies, the following checks are performed: 4

- For a subprogram call, a check is made that the body of the subprogram is already elaborated. The check that the body of a subprogram has been elaborated is made *after* the evaluation of all of the actual parameters of a call.[16] 5
- ♦ 6

The exception PROGRAM_ERROR is raised if this check fails. 8

If a subprogram declaration ♦ or a package declaration is a declarative item of a given declarative part, then the body (if there is one) of the program unit declared by the declarative item must itself be a declarative item of this declarative part (and must appear later). ♦ 9

References: inner_declaration 3.1, ♦ program_error exception 11.1, scope 8.2, subprogram call 6.4, type 3.3, visibility 8.3 10

Elaboration of declarations: 3.1, component declaration 3.7, deferred constant declaration 7.4.3, enumeration literal specification 3.5.1, ♦ loop parameter specification 5.5, number declaration 3.2.2, object declaration 3.2.1, package declaration 7.2, parameter specification 6.1, private type declaration 7.4.1, renaming declaration 8.5, subprogram declaration 6.1, subtype declaration 3.3.2 11

---

[16]**IMPLEMENTATION REQUIREMENT**. This eliminates one source of IODs. See [AI-00406] for the Ada requirement that no such ordering be specified.

Elaboration of type definitions: 3.3.1, array type definition 3.6, enumeration type definition 3.5.1, integer type definition 3.5.4, record type definition 3.7 12

Elaboration of other constructs: context clause 10.1, compilation unit 10.1, library unit 10.5, package body 7.1, subprogram body 6.3, use clause 8.4, with clause 10.1.1 13

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
3.09	(05)	BI	WJ	0180/07	86-07-23	Elaboration checks for INTERFACE subprograms
3.09	(05)	BI	WJ	0406/05	87-08-06	Evaluating parameters of a call before raising PROGRAM_ERROR
3.09	(06)	BI	WJ	0149/09	87-02-23	Activating a task before elaboration of its body
3.09	(08)	BI	WJ	0430/05	88-05-23	Using an enumeration literal does not raise PROGRAM_ERROR

Chapter 4
NAMES AND EXPRESSIONS

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

4.1 Names

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

```
name ::= simple_name                                          2
   | character_literal  | ♦
   | indexed_component  | ♦
   | selected_component | attribute

simple_name ::= identifier

prefix ::= name | function_call
```

A simple name for an entity is either the identifier associated with the entity by its declaration, or another identifier associated with the entity by a renaming declaration. 3

Certain forms of name (indexed and selected components ♦ and attributes) include a *prefix* that is either a name or a function call. ♦ 4

If the prefix of a name is a function call, then the name denotes a component ♦ or an attribute ♦ of the result of the function call ♦. 5

A prefix is said to be *appropriate for a type* in ♦ the following case: 6

- The type of the prefix is the type considered. 7
- ♦ 8

The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is a simple name or a character literal ♦. 9

The evaluation of a name that has a prefix includes the evaluation of the prefix, that is, of the corresponding name or function call. ♦ 10

Examples of simple names: 11

```
♦
LIMIT          -- the simple name of a constant          (see 3.2.1)
♦
BOARD          -- the simple name of an array variable   (see 3.6.1)
TABLE          -- the simple name of a type              (see 3.6)
INCREMENT      -- the simple name of a function          (see 6.1)
♦
```

References: attribute 4.1.4, belong to a type 3.3, character literal 2.5, component 3.3, constraint_error exception 11.1, declaration 3.1, designate 3.8, designated type 3.8, entity 3.1, evaluation 4.5, formal parameter 6.1, function call 6.4, identifier 2.3, indexed component 4.1.1, mode 6.1, object 3.2.1, ♦ raising of exceptions 11, renaming declarations 8.5, selected component 4.1.3, subcomponent 3.3, type 3.3 12

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.01	(10)	BI	RE	0676/00	88-12-21	A mistake?

4.1.1 Indexed Components

An indexed component denotes ♦ a component of an array ♦. 1

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

♦ The prefix must be appropriate for an array type. The expressions specify the index values for the component; there must be one such expression for each index position of the array type. ♦ 3

Each expression must be of the type of the corresponding index. For the evaluation of an indexed component, the prefix is evaluated first, followed by the expressions (left to right).[17] The exception CONSTRAINT_ERROR is raised if an index value does not belong to the range of the corresponding index of the prefixing array ♦. 4

Examples of indexed components: 5

```
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 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). ♦ 6

References: appropriate for a type 4.1, array type 3.6, component 3.3, component of an array 3.6, constraint_error exception 11.1, dimension 3.6, evaluation 4.5, expression 4.4, function call 6.4, in some order 1.6, index 3.6, name 4.1, prefix 4.1, raising of exceptions 11, returned value 5.8 6.5 7

---

[17] **IMPLEMENTATION REQUIREMENT**.

AI Crossreferences:

Section	Class	Status	AI-0	Date	Description
4.01.01	(04)	BI RE	0585/00	88-09-02	discriminant change after prefix evaluation

4.1.2 Slices: Removed

4.1.3 Selected Components

Selected components are used to denote record components ♦; they are also used as *expanded names* as described below.

selected_component ::= prefix.selector

selector ::= simple_name

The following ♦ forms of selected components are used to denote ♦ a record component ♦:

(a) Omitted

(b) A component of a record:

The selector must be a simple name denoting a component of a record object or value. The prefix must be appropriate for the type of this object or value.

(c) Omitted

(d) Omitted

A selected component of the remaining form is called an *expanded name*. In this case the selector must be ♦ a simple name ♦. A function call is not allowed as the prefix of an expanded name. An expanded name can denote:

(e) An entity declared in the visible part of a package:

The prefix must denote the package. The selector must be the simple name ♦ of the entity.

(f) Omitted

If, according to the visibility rules, there is at least one possible interpretation of the prefix of a selected component as the name of an enclosing subprogram,[18] then the expanded name is in error. In particular, no interpretations of the prefix as a function call are then considered.

The evaluation of a name that is a selected component includes the evaluation of the prefix.

Examples of selected components:

```
TOMORROW.MONTH                -- a record component    (see 3.7)
NEXT_CAR.OWNER                -- a record component    (see 3.7)
NEXT_PERSON.VEHICLE.NUMBER    -- a record component    (see 3.7)
```

Examples of expanded names:

[18]We take ‘‘name of an enclosing subprogram’’ to only include the names that are literally present and to exclude any renaming of these that introduce alternative names [see AI-0119].

```
TABLE_MANAGER.INSERT      -- a procedure of the visible part of a package  (see 7.5)
♦
STANDARD.BOOLEAN          -- the name of a predefined type                 (see 8.6 and C)
```

Note:

For a record with components that are other records, the above rules imply that the simple name must be given at each level for the name of a subcomponent. For example, the name NEXT_CAR.OWNER.BIRTH.MONTH cannot be shortened (NEXT_CAR.OWNER.MONTH is not allowed). 23

References: ♦ appropriate for a type 4.1, ♦ body of a program unit 3.9, character literal 2.5, component of a record 3.7, constraint_error exception 11.1, declaration 3.1, designate 3.8, entity 3.1, function call 6.4, index 3.6, ♦ object 3.2.1, occur immediately within 8.1, operator 4.5, ♦ overloading 8.3, package 7, predefined type C, prefix 4.1, procedure body 6.3, program unit 6, raising of exceptions 11, record 3.7, record component 3.7, renaming declaration 8.5, reserved word 2.9, simple name 4.1, subprogram 6, variable 3.7.3, visibility 8.3, visible part 3.7.3 24

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.01.03	(15)	BI	WA	0504/03	88-06-16	Expanded names with a renamed prefix in generic packages
4.01.03	(15)	BI	WJ	0016/10	86-08-05	Using a renamed package prefix inside a package
4.01.03	(15)	BI	WJ	0187/06	87-09-12	Using a name decl by a renaming decl as an expanded name selector
4.01.03	(15)	BI	WJ	0412/06	88-05-23	Expanded names for generic formal parameters
4.01.03	(15)	CR	WJ	0468/04	87-09-12	Correction to 0187/04 discussion
4.01.03	(17)	BI	RE	0593/00	88-10-05	Visibility of accept statements
4.01.03	(17)	BI	WI	0119/02	88-03-28	The prefix of an expanded name
4.01.03	(18)	BI	WA	0504/03	88-06-16	Expanded names with a renamed prefix in generic packages
4.01.03	(18)	BI	WI	0119/02	88-03-28	The prefix of an expanded name
4.01.03	(18)	BI	WJ	0016/10	86-08-05	Using a renamed package prefix inside a package
4.01.03	(18)	BI	WJ	0412/06	88-05-23	Expanded names for generic formal parameters
4.01.03	(19)	BI	WI	0119/02	88-03-28	The prefix of an expanded name

4.1.4 Attributes

An attribute denotes a basic operation of an entity given by a prefix. 1

attribute ::= prefix’attribute_designator 2

attribute_designator ::= simple_name [(*universal_static*_expression)]

The applicable attribute designators depend on the prefix. An attribute can be a basic operation delivering a value; alternatively it can be a function, a type, or a range. The meaning of the prefix of an attribute must be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute. 3

The attributes defined by the language are summarized in Annex A. In addition, an implementation may provide implementation-defined attributes; their description must be given in Appendix F. The attribute designator of any implementation-defined attribute must not be the same as that of any language-defined attribute. 4

The evaluation of a name that is an attribute has an effect that depends on the specific attribute. The result of this evaluation may be a value, a type, or a function to be applied. In case the prefix is appropriate for an array type, the evaluation of a name includes the evaluation of the 5

prefix.

Notes:

The attribute designator ♦ RANGE has the same identifier as a reserved word. However, no confusion is possible since an attribute designator is always preceded by an apostrophe. The only predefined attribute designators that have a universal expression are those for certain operations of array types (see 3.6.2). 6

Examples of attributes: 7

```
COLOR'FIRST              -- minimum value of the enumeration type COLOR   (see 3.3.1 and 3.5)
RAINBOW'BASE'FIRST       -- same as COLOR'FIRST                           (see 3.3.2 and 3.3.3)
♦
BOARD'LAST(2)            -- upper bound of the second dimension of BOARD  (see 3.6.1 and 3.6.2)
BOARD'RANGE(1)           -- index range of the first dimension of BOARD   (see 3.6.1 and 3.6.2)
♦
```

References: appropriate for a type 4.1, basic operation 3.3.3, declared entity 3.1, name 4.1, prefix 4.1, reserved word 2.9, simple name 4.1, static expression 4.9, type 3.3, universal expression 4.10 8

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.01.04	(03)	BI	WI	0188/01	84-06-12	Implementation Defined Attributes
4.01.04	(03)	na	na	0061/00	85-08-01	[combined with 0015]
4.01.04	(03)	ra	WJ	0015/12	86-12-01	When the prefix of 'ADDRESS contains a function name
4.01.04	(04)	BI	CE	0009/04	88-09-02	Implementation-defined names cannot be reserved words

4.2 Literals

A literal is either a numeric literal, an enumeration literal, ♦ or a string literal. The evaluation of a literal yields the corresponding value. 1

Numeric literals are the literals of the type *universal_integer* ♦. Enumeration literals include character literals and yield values of the corresponding enumeration types. ♦ 2

A string literal is a basic operation that combines a sequence of characters into a value of a one-dimensional array of a character type; the bounds of this array are determined according to the rules for positional array aggregates (see 4.3.2). For a null string literal, the upper bound is the predecessor, as given by the PRED attribute, of the lower bound. The evaluation of a null string literal raises the exception CONSTRAINT_ERROR if the lower bound does not have a predecessor (see 3.5.5). 3

The type of a string literal ♦ must be determinable solely from the context in which this literal appears, excluding the literal itself, but using the fact that ♦ a string literal is a value of a one-dimensional array type whose component type is a character type. 4

The character literals corresponding to the graphic characters contained within a string literal must be visible at the place of the string literal (although these characters themselves are not used to determine the type of the string literal). 5

Examples: 6

♦
```
1_345              -- an integer literal
CLUBS              -- an enumeration literal
'A'                -- a character literal
"SOME TEXT"        -- a string literal
```

References: ♦ aggregate 4.3, array 3.6, array bound 3.6, array type 3.6, character literal 2.5, character type 3.5.2, component type 3.3, constraint_error exception 11.1, designate 3.8, dimension 3.6, enumeration literal 3.5.1, graphic character 2.1, integer literal 2.4, null literal 3.8, numeric literal 2.4, object 3.2.1, string literal 2.6, type 3.3, universal_integer type 3.5.4, visibility 8.3 7

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.02	(03)	BI	WA	0472/04	88-06-16	Graphic characters not in a string literal's component type
4.02	(05)	ST	RE	0390/00	85-09-16	Visibility of character literals.
4.02	(05)	ST	RE	0420/03	88-11-08	Allow 256 values for type CHARACTER

4.3 Aggregates

An aggregate is a basic operation that combines component values into a composite value of a record or array type. 1

aggregate ::= 2
  (component_association {, component_association})

component_association ::=
  [choice {| choice} =>] expression

choice[19] ::= simple_expression | discrete_range
    | *component*_simple_name | **others**

Each component association associates an expression with components (possibly none). A component association is said to be *named* if the components are specified explicitly by choices; it is otherwise said to be *positional*. For a positional association, the (single) component is implicitly specified by position, in the order of the corresponding component declarations for record components, in index order for array components. The only choices allowed in an aggregate are *component*_simple_name or **others**. The other possibilities support the use of choice in case statements. 3

Named associations can be given in any order. Positional and named associations cannot be used in the same aggregate. Aggregates containing a single component association must always be given in named notation. (Consider this narrative requirement to be a syntax rule for purposes of overload resolution [AI-00157].) Specific rules concerning component associations exist for record aggregates and array aggregates. 4

♦ A choice that is a component simple name is only allowed in a record aggregate. ♦ The choice **others** is only allowed in a component association if the association ♦ has this single choice and the containing aggregate has this single component association; it specifies all components, which must be of the same type. 5

[19] Moved from 3.7.3, which has been removed.

Each component of the value defined by an aggregate must be represented once and only once in the aggregate and must be a defined component of the type [AI-00309]. Hence each aggregate must be complete and a given component is not allowed to be specified by more than one choice. 6

The type of an aggregate must be determinable solely from the context in which the aggregate appears, excluding the aggregate itself, but using the fact that this type must be composite ♦. The type of an aggregate in turn determines the required type for each of its components. 7

Notes:

The above rule implies that the determination of the type of an aggregate cannot use any information from within the aggregate. In particular, this determination cannot use the type of the expression of a component association, or the form or the type of a choice. An aggregate can always be distinguished from an expression enclosed by parentheses: this is a consequence of the fact that named notation is required for an aggregate with a single component. 8

References: array aggregate 4.3.2, array type 3.6, basic operation 3.3.3, component 3.3, composite type 3.3, composite value 3.3, discrete range 3.6, expression 4.4, index 3.6, ♦ primary 4.4, record aggregate 4.3.1, record type 3.7, simple expression 4.4, simple name 4.1, type 3.3 9

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.03	(06)	BI	WI	0309/02	88-07-06	Aggregates with choices outside the aggregate's subtype
4.03	(06)	co	WJ	0293/05	87-06-18	Null **others** choice for array aggregates
4.03	(06)	na	na	0491/01	87-04-19	[Combined with 0309]
4.03	(06)	ra	WJ	0169/06	86-07-23	Legality of incomplete null multidimensional array aggregates

4.3.1 Record Aggregates

If the type of an aggregate is a record type, the component names given as choices must denote all components of the record type. If the choice **others** is given as a choice of a record aggregate, it must represent all components and there must be at least one component. A component association with the choice **others** or with more than one choice is only allowed if the represented components are all of the same type. The expression of a component association must have the type of the associated record components. 1

♦ 2

For the evaluation of a record aggregate, the expressions given in the component associations are evaluated left to right.[20] The expression of a named association is evaluated once for each associated component. A check is made that the value of each subcomponent of the aggregate belongs to the subtype of this subcomponent. The exception CONSTRAINT_ERROR is raised if this check fails. 3

Example of a record aggregate with positional associations: 4

```
(4, JULY, 1776)          -- see 3.7
```

Examples of record aggregates with named associations: 5

[20] **IMPLEMENTATION REQUIREMENT**.

```
(DAY => 4, MONTH => JULY, YEAR => 1776)
(MONTH => JULY, DAY => 4, YEAR => 1776)
```
♦

Example of component association with several choices: 6

```
(MONTH => JULY, DAY|YEAR => 0)      -- see 3.7
```

References: aggregate 4.3, choice 4.3, component association 4.3, component name 3.7, constraint 3.3, constraint_error exception 11.1, evaluate 4.5, expression 4.4, in some order 1.6, program 10, raising of exceptions 11, record component 3.7, record type 3.7, satisfy 3.3, static expression 4.9, subcomponent 3.3, subtype 3.3.2, type 3.3 8

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.03.01	(01)	BI	WI	0309/02	88-07-06	Aggregates with choices outside the aggregate's subtype
4.03.01	(01)	BI	WJ	0244/04	87-01-20	Record aggregates with multiple choices in a component association
4.03.01	(01)	ST	RE	0681/00	88-12-21	Can't declare a constant of a 'null' record type.
4.03.01	(03)	BI	WA	0189/06	88-11-04	Order of evaluation of components in a record aggregate

4.3.2 Array Aggregates

If the type of an aggregate is a one-dimensional array type, then the only allowed choice is **others**, and the expression of each component association must be of the component type. 1

If the type of an aggregate is a multidimensional array type, an n-dimensional aggregate is written as a one-dimensional aggregate, in which the expression specified for each component association is itself written as an (n-1)-dimensional aggregate which is called a *subaggregate*; the index subtype of the one-dimensional aggregate is given by the first index position of the array type. The same rule is used to write a subaggregate if it is again multidimensional, using successive index positions. A string literal is allowed in a multidimensional aggregate at the place of a one-dimensional array of a character type. In what follows, the rules concerning array aggregates are formulated in terms of one-dimensional aggregates. 2

♦ The component associations of an array aggregate must be either all positional or a single association with the single choice **others**. ♦ An **others** choice is static if the applicable index constraint is static. 3

The bounds of an array aggregate that has an **others** choice are determined by the applicable index constraint. An **others** choice is only allowed if the aggregate appears in one of the following contexts (which defines the applicable index constraint): 4

(a) The aggregate is an actual parameter, ♦ the result expression of a function, or the expression that follows an assignment compound delimiter. Moreover, the subtype of the corresponding formal parameter, ♦ function result, or object is a constrained array subtype. An object is a constrained array subtype if it declared to be one or if it is a formal that is defined to be an unconstrained array type. 5

(b) The aggregate is the operand of a qualified expression whose type mark denotes a constrained array subtype. 7

(c) The aggregate is not a subaggregate and is the expression of a component association of an enclosing (array or record) aggregate. ♦ 8

(d) The aggregate is a subaggregate of a multidimensional array aggregate that is in one of the previous three contexts [AI-00177]. 8a

The bounds of an array aggregate that does not have an **others** choice are determined as follows. ♦ For a positional aggregate, the lower bound is determined by the applicable index constraint if the aggregate appears in one of the contexts (a) through (d); otherwise, the lower bound is given by S'FIRST where S is the index subtype; in either case, the upper bound is determined by the number of components. 9

The evaluation of an array aggregate that is not a subaggregate proceeds in one step. The choices of this aggregate and of its subaggregates, if any, are not evaluated, since the only possible choice for an array aggregate is OTHERS. The expressions of the component associations of the array aggregate are evaluated left to right[21]; the expression of a named association is evaluated once for each associated component. The evaluation of a subaggregate consists of this step. 10

For the evaluation of an aggregate ♦ a check is made that ♦ the value of each subcomponent of the aggregate belongs to the subtype of this subcomponent. For an n-dimensional multidimensional aggregate, a check is made that all (n-1)-dimensional subaggregates have the same bounds. The exception CONSTRAINT_ERROR is raised if any of these checks fails. CONSTRAINT_ERROR is also raised if the bounds of a positional aggregate do not belong to the corresponding index subtype [AI-00019]. These checks are all made after the component expressions have been evaluated.[22] 11

Note:

The allowed contexts for an array aggregate including an **others** choice are such that the bounds of such an aggregate are always known from the context. 12

Examples of array aggregates with positional associations: 13

```
(7, 9, 5, 1, 3, 2, 4, 8, 6, 0)
♦

♦
```
14

Examples of two-dimensional array aggregates: 15

```
-- Two aggregates for values of type MATRIX (see 3.6):

((1, 1, 1), (2, 2, 2))
(others => (1, 1, 1))
♦
```

Examples of aggregates as initial values: 16

```
A : TABLE := (7, 9, 5, 1, 3, 2, 4, 8, 6, 0);          -- A(1)=7, A(10)=0
B : TABLE := 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"
```

References: actual parameter 6.4.1, aggregate 4.3, array type 3.6, assignment compound delimiter 5.2, choice 4.3, component 3.3, component association 4.3, component type 3.3, constrained array subtype 3.6, 17

---

[21] **IMPLEMENTATION REQUIREMENT.**

[22] **IMPLEMENTATION REQUIREMENT**.

constraint 3.3, constraint_error exception 11.1, dimension 3.6, evaluate 4.5, expression 4.4, formal parameter 6.1, function 6.5, in some order 1.6, index constraint 3.6.1, index range 3.6, index subtype 3.6, index type 3.6, named component association 4.3, null array 3.6.1, object 3.2, positional component association 4.3, qualified expression 4.7, raising of exceptions 11, static expression 4.9, subcomponent 3.3, type 3.3

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.03.02	(02)	co	WA	0273/04	88-06-17	The bounds of a string literal in a multidimensional aggregate
4.03.02	(02)	ra	WA	0017/05	88-06-17	Use of a string value in a multidimensional aggregate
4.03.02	(03)	BI	WA	0414/04	88-06-17	Null discrete range as an array aggregate choice
4.03.02	(03)	BI	WJ	0190/05	86-07-23	A static expression cannot have a generic formal type
4.03.02	(03)	BI	WJ	0310/04	86-07-23	OTHERS choices and static index constraints
4.03.02	(04)	na	na	0415/01	87-01-20	[combined with 0244]
4.03.02	(05)	BI	CE	0568/02	88-11-20	Unconstrained array objects and ***others*** choices
4.03.02	(06)	BI	WI	0473/01	87-04-16	Named associations for default array aggregates
4.03.02	(08)	BI	WJ	0177/04	86-07-23	Use of others in a multidimensional aggregate
4.03.02	(11)	BI	WJ	0019/07	86-07-23	Checking for too many components in positional aggregates
4.03.02	(11)	BI	WJ	0313/03	86-07-23	Non-null bounds belong to the index subtype
4.03.02	(11)	co	WA	0176/04	88-11-04	Bounds of subaggregates in null arrays
4.03.02	(11)	co	WA	0437/01	88-06-17	Evaluation of multi-dimensional array aggregates
4.03.02	(11)	ra	WJ	0018/06	86-07-23	Checking aggregate index and subcomponent values
4.03.02	(11)	ra	WJ	0265/05	87-06-18	Index subtype of an array aggregate

4.4 Expressions

An expression is a formula that defines the computation of a value. 1

expression ::= 2
    relation {**and** relation} | relation {**and then** relation}
  | relation {**or** relation} | relation {**or else** relation}
  | relation {**xor** relation}

relation ::=
    simple_expression [relational_operator simple_expression]
  | simple_expression [**not**] **in** range
  | simple_expression [**not**] **in** type_mark

simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | **abs** primary | **not** primary

primary ::=
    numeric_literal | ♦ | aggregate | string_literal | name | ♦
  | function_call | type_conversion | qualified_expression | (expression)

Each primary has a value and a type. The only names allowed as primaries are named numbers; attributes that yield values; and names denoting ♦ values. ♦ 3

The possible types of an expression depend only on the type of its constituents and on the operators applied except for string literals and aggregates; for an overloaded constituent or operator, the determination of the constituent type, or the identification of the appropriate operator, depends on the context. The types of string literals and aggregates must be determinable solely from the context in which they appear. For each predefined operator, the operand and result types are given in section 4

4.5.

Examples of primaries: 5

```
♦
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: 6

```
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)
```

References: aggregate 4.3, ♦ array aggregate 4.3.2, attribute 4.1.4, binary adding operator 4.5 4.5.3, context of overload resolution 8.7, exponentiating operator 4.5 4.5.6, function call 6.4, multiplying operator 4.5 4.5.5, name 4.1, named number 3.2, numeric literal 2.4, object 3.2, operator 4.5, overloading 8.3, overloading an operator 6.7, qualified expression 4.7, range 3.5, ♦ relation 4.5.1, relational operator 4.5 4.5.2, result type 6.1, string literal 2.6, type 3.3, type conversion 4.6, type mark 3.3.2, unary adding operator 4.5 4.5.4, variable 3.2.1 7

4.5 Operators and Expression Evaluation

The language defines the following six classes of operators. ♦ They are given in the order of increasing precedence. 1

```
logical_operator            ::= and | or | xor                    2

relational_operator         ::= = | /= | < | <= | > | >=

binary_adding_operator      ::= + | - | &

unary_adding_operator       ::= + | -

multiplying_operator        ::= * | / | mod | rem

highest_precedence_operator ::= ** | abs | not
```

The short-circuit control forms **and then** and **or else** have the same precedence as logical operators. The membership tests **in** and **not in** have the same precedence as relational operators. 3

For a term, simple expression, relation, or expression, operators of higher precedence are associated with their operands before operators of lower precedence. In this case, for a sequence of operators of the same precedence level, the operators are associated in textual order from left to right; parentheses can be used to impose specific associations. 4

The operands of a factor, of a term, of a simple expression, or of a relation, and the operands of an expression that does not contain a short-circuit control form, are evaluated left to right[23] (but before application of the corresponding operator). The right operand of a short-circuit control form is evaluated if and only if the left operand has a certain value (see 4.5.1). It is poor programming style for the evaluation of the operands of an expression to modify variables that are mutually accessible by any pair of operands. 5

For each form of type declaration, certain of the above operators are *predefined*, that is, they are implicitly declared by the type declaration. For each such implicit operator declaration, the names of the parameters are LEFT and RIGHT for binary operators; the single parameter is called RIGHT for unary adding operators and for the unary operators **abs** and **not**. The effect of the predefined operators is explained in subsections 4.5.1 through 4.5.7 and in 4.10 for universal expressions [AI-00689]. 6

The predefined operations on integer types either yield the mathematically correct result or raise the exception CONSTRAINT_ERROR.[24] A predefined operation that delivers a result of an integer type (other than *universal_integer*) can only raise the exception CONSTRAINT_ERROR if the mathematical result is not a value of the type. ♦ 7

Examples of precedence: 8

```
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
```

*References: constraint_*error exception 11.1, designator 6.1, expression 4.4, factor 4.4, implicit declaration 3.1, in some order 1.6, integer type 3.5.4, membership test 4.5.2, name 4.1, overloading 6.6 8.7, raising of an exception 11, range 3.5, relation 4.4, short-circuit control form 4.5 4.5.1, simple expression 4.4, term 4.4, type 3.3, type declaration 3.3.1, universal_integer type 3.5.4 9

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.05	(07)	NB	WJ	0387/05	87-02-23	Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR

[23] **IMPLEMENTATION REQUIREMENT**.

[24] AI-00387.

4.5.1 Logical Operators and Short-circuit Control Forms

The following logical operators are predefined for any boolean type and any one-dimensional array type whose components are of a boolean type; in either case the two operands have the same type. 1

Operator	*Operation*	*Operand type*	*Result type*	2
and	conjunction	any boolean type array of boolean components	same boolean type same array type[25]	
or	inclusive disjunction	any boolean type array of boolean components	same boolean type same array type	
xor	exclusive disjunction	any boolean type array of boolean components	same boolean type same array type	

The operations on arrays are performed on a component-by-component basis on matching components, if any (as for equality, see 4.5.2). The bounds of the resulting array are those of the left operand. A check is made that for each component of the left operand there is a matching component of the right operand, and vice versa. The exception CONSTRAINT_ERROR is raised if this check fails. When the predefined boolean operators (and **not**) deliver a result having an array type, CONSTRAINT_ERROR is raised if any component of the result does not belong to the component subtype [AI-00535]. 3

The short-circuit control forms **and then** and **or else** are defined for two operands of a boolean type and deliver a result of the same type. The left operand of a short-circuit control form is always evaluated first. If the left operand of an expression with the control form **and then** evaluates to FALSE, the right operand is not evaluated and the value of the expression is FALSE. If the left operand of an expression with the control form **or else** evaluates to TRUE, the right operand is not evaluated and the value of the expression is TRUE. If both operands are evaluated, **and then** delivers the same result as **and**, and **or else** delivers the same result as **or**. 4

♦ The conventional meaning of the logical operators is given by the following truth table:[26] 5

A	B	(A **and** B)	(A **or** B)	(A **xor** B)	6
TRUE	TRUE	TRUE	TRUE	FALSE	
TRUE	FALSE	FALSE	TRUE	TRUE	
FALSE	TRUE	FALSE	TRUE	TRUE	
FALSE	FALSE	FALSE	FALSE	FALSE	

Examples of logical operators: 7

```
SUNNY or WARM
```
♦

Examples of short-circuit control forms: 8

```
NEXT_PERSON.AGE /= 0 and then 25 / NEXT_PERSON.AGE < 1   --   see 3.7
N = 0 or else A(N) = HIT_VALUE
```

---

[25] We had intended to delete these overloadings. Unfortunately that would transform some *ambiguous* Ada programs into *unambiguous* AVA programs. We could make such occurrences illegal if an operand could be interpreted as an array of booleans. But we need overloading anyway.

[26] This was in a note. It now has definitional force.

References: boolean type 3.5.3, bound of an index range 3.6.1, false boolean value 3.5.3, operation 3.3, operator 4.5, predefined operator 4.5, true boolean value 3.5.3, type 3.3 9

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.05.01	(01)	ra	RE	0474/01	86-11-09	Definition of predefined operators.
4.05.01	(03)	BI	WA	0535/03	88-11-04	Boolean operators producing out of range results
4.05.01	(03)	BI	WJ	0426/05	87-06-18	Operations on undefined array values
4.05.01	(03)	ra	WJ	0431/05	88-05-23	Predefined logical operators for boolean arrays
4.05.01	(04)	BI	WA	0391/06	88-06-16	Order of evaluation for short-circuit control forms

4.5.2 Relational Operators and Membership Tests

The equality and inequality operators are predefined for every type. 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 any scalar type, and for any discrete array type, that is, a one-dimensional array type whose components are of a discrete type. The operands of each predefined relational operator have the same type. The result type is the predefined type BOOLEAN. 1

The relational operators have their conventional meaning: the result is equal to TRUE if the corresponding relation is satisfied; the result is FALSE otherwise. The inequality operator gives the complementary result to the equality operator: FALSE if equal, TRUE if not equal. 2

Operator	*Operation*	*Operand type*	*Result type*
= /=	equality and inequality	any type	BOOLEAN
< <= > >=	test for ordering	any scalar type	BOOLEAN
		discrete array type	BOOLEAN

3

Equality for the discrete types is equality of the values. ♦ 4

For two array values or two record values of the same type, the left operand is equal to the right operand if and only if for each component of the left operand there is a *matching component* of the right operand and vice versa; and the values of matching components are equal, as given by the predefined equality operator for the component type. In particular, two null arrays of the same type are always equal ♦. 5

For comparing two records of the same type, *matching components* are those which have the same component identifier. 6

For comparing 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. For comparing two multidimensional arrays, matching components are those whose index values match in successive index positions. 7

♦ 8

The ordering operators <, <=, >, and >= that are defined for discrete array types 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 9

right (the tail consists of the remaining components beyond the first and can be null).

The membership tests **in** and **not in** are predefined for all types.[27] The result type is the predefined type BOOLEAN. For a membership test with a range, the simple expression and the bounds of the range must be of the same scalar type; for a membership test with a type mark, the type of the simple expression must be the base type of the type mark. The evaluation of the membership test **in** yields the result TRUE if the value of the simple expression is within the given range, or if this value belongs to the subtype denoted by the given type mark; otherwise this evaluation yields the result FALSE (for a value of a real type, see 4.5.7). The membership test **not in** gives the complementary result to the membership test **in**. 10

Examples: 11

```
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)
♦
```

Notes:

No exception is ever raised by a predefined relational operator or by a membership test, but an exception can be raised by the evaluation of the operands. 12

♦ Two nonnull arrays have matching components if and only if the value of the attribute LENGTH(N) for each index position N is the same for both. 13

References: array type 3.6, base type 3.3, belong to a subtype 3.3, boolean predefined type 3.5.3, bound of a range 3.5, component 3.3, component identifier 3.7, component type 3.3, composite type 3.3, designate 3.8, dimension 3.6, discrete type 3.5, evaluation 4.5, exception 11, index 3.6, index range 3.6, ♦ null array 3.6.1, ♦ object 3.2.1, operation 3.3, operator 4.5, predefined operator 4.5, raising of exceptions 11, range 3.5, record type 3.7, scalar type 3.5, simple expression 4.4, subcomponent 3.3, successor 3.5.5, type 3.3, type mark 3.3.2 14

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.05.02	(10)	co	RE	0207/00	84-03-13	Intended types

4.5.3 Binary Adding Operators

The binary adding operators + and - are predefined for any numeric type and have their conventional meaning. The catenation operators & are predefined for any one-dimensional array type ♦. 1

Operator	*Operation*	*Left operand type*	*Right operand type*	*Result type*	2
+	addition	any numeric type	same numeric type	same numeric type	

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

-	subtraction	any numeric type	same numeric type	same numeric type
&	catenation	any array type	same array type	same array type
		any array type	the component type	same array type[28]
		the component type	any array type	same array type
		the component type	the component type	any array type

♦

If both operands are one-dimensional arrays, the result of the catenation 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. The lower bound of this result is the lower bound of the left operand, unless the left operand is a null array, in which case the result of the catenation is the right operand. 4

If either operand is of the component type of an array type, the result of the catenation is given by the above rules, using in place of this operand an array having this operand as its only component and having the lower bound of the index subtype of the array type as its lower bound. 4

The exception CONSTRAINT_ERROR is raised by catenation if the upper bound of the result exceeds the range of the index subtype, unless the result is a null array. ♦ 5

Examples: 6

♦
```
"A" & "BCD"  --  catenation of two string literals
'A' & "BCD"  --  catenation of a character literal and a string literal
'A' & 'A'    --  catenation of two character literals
```

References: array type 3.6, character literal 2.5, component type 3.3, constraint_error exception 11.1, dimension 3.6, index subtype 3.6, length of an array 3.6.2, ♦ null array 3.6.1, numeric type 3.5, operation 3.3, operator 4.5, predefined operator 4.5, raising of exceptions 11, range of an index subtype 3.6.1, string literal 2.6, type 3.3 7

4.5.4 Unary Adding Operators

The unary adding operators + and - are predefined for any numeric type and have their conventional meaning. For each of these operators, the operand and the result have the same base type. 1

Operator	*Operation*	*Operand type*	*Result type*
+	identity	any numeric type	same numeric type
-	negation	any numeric type	same numeric type

2

References: numeric type 3.5, operation 3.3, operator 4.5, predefined operator 4.5, type 3.3 3

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

4.5.5 Multiplying Operators

The operators * and / are predefined for any integer ♦ type and have their conventional meaning; the operators **mod** and **rem** are predefined for any integer type. For each of these operators, the operands and the result have the same base type. ♦ 1

Operator	*Operation*	*Operand type*	*Result type*
*	multiplication	any integer type ♦	same integer type ♦
/	integer division ♦	any integer type ♦	same integer type ♦
mod	modulus	any integer type	same integer type
rem	remainder	any integer type	same integer type

2

Integer division and remainder are defined by the relation 3

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. Integer division satisfies the identity 4

(-A)/B = -(A/B) = A/(-B)

The result of the modulus operation is such that (A **mod** B) has the sign of B and an absolute value less than the absolute value of B; in addition, for some integer value N, this result must satisfy the relation 5

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

♦ 6

The exception CONSTRAINT_ERROR[29] is raised by integer division, **rem**, and **mod** if the right operand is zero. 12

Examples: 13

```
I : INTEGER := 1;
J : INTEGER := 2;
K : INTEGER := 3;
```

♦

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

♦

Notes:

For positive A and B, A/B is the quotient and A **rem** B is the remainder when A is divided by B. The 14

[29] AI-00387.

following relations are satisfied by the **rem** operator:

A **rem** (-B) = A **rem** B
(-A) **rem** B = -(A **rem** B)

For any integer K, the following identity holds: 15

A **mod** B = (A + K*B) **mod** B

The relations between integer division, remainder, and modulus are illustrated by the following table: 16

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
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

References: actual parameter 6.4.1, base type 3.3, *constraint*_error exception 11.1, declaration 3.1, ♦ integer type 3.5.4, numeric type 3.5, predefined operator 4.5, raising of exceptions 11, renaming declaration 8.5, standard predefined package 8.6, type conversion 4.6 17

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.05.05	(08)	BI	WJ	0475/05	88-05-23	Multiplication of fixed point values by negative integers
4.05.05	(10)	ra	WJ	0020/07	86-07-23	Real literals with fixed point multiplication and division
4.05.05	(10)	ra	WJ	0376/04	86-12-01	Universal real operands with fixed point multiply and /
4.05.05	(10)	ST	RE	0262/01	87-02-10	Real literals with fixed point multiplication and division
4.05.05	(11)	BI	RE	0522/00	87-02-10	Counter-Productive Accuracy of Universal Fixed
4.05.05	(11)	ra	WJ	0235/05	88-05-23	Redundant parentheses enclosing *universal_fixed* expressions do
4.05.05	(12)	NB	WJ	0387/05	87-02-23	Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR

4.5.6 Highest Precedence Operators

The highest precedence unary operator **abs** is predefined for any numeric type. The highest precedence unary operator **not** is predefined for any boolean type and any one-dimensional array type whose components have a boolean type. 1

Operator	*Operation*	*Operand type*	*Result type*
abs	absolute value	any numeric type	same numeric type
not	logical negation	any boolean type array of boolean components	same boolean type same array type

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). 3

The highest precedence *exponentiating* operator ** is predefined for each integer type ♦. The right 4

operand, called the exponent, is of the predefined type INTEGER.

Operator	*Operation*	*Left operand type*	*Right operand type*	*Result type*
**	exponentiation	any integer type ♦	INTEGER	same as left

Exponentiation with a positive exponent is equivalent to repeated multiplication of the left operand by itself, as indicated by the exponent and from left to right. ♦ Exponentiation by a zero exponent delivers the value one. ♦ Exponentiation of an integer raises the exception CONSTRAINT_ERROR for a negative exponent.

References: array type 3.6, boolean type 3.5.3, bound of an array 3.6.1, component of an array 3.6, constraint_error exception 11.1, dimensionality 3.6, index 3.6, integer type 3.5.4, multiplication operation 4.5.5, predefined operator 4.5, raising of exceptions 11

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.05.06	(00)	ST	RE	0460/00	86-10-02	Allow non-integral powers for exponentiation
4.05.06	(03)	BI	WA	0535/03	88-11-04	Boolean operators producing out of range results
4.05.06	(06)	ra	CA	0476/03	88-05-02	Model interval for exponentiation with a negative exponent
4.05.06	(06)	ra	WJ	0137/05	86-07-23	Exponentiation with floating point operand

4.5.7 Accuracy of Operations with Real Operands: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.05.07	(00)	BI	WJ	0407/06	88-05-23	The operations of a subtype with reduced accuracy
4.05.07	(04)	BI	WJ	0516/05	88-05-23	The safe interval for a fixed/integer result
4.05.07	(07)	BI	CA	0174/07	88-10-03	T'FIRST and T'LAST for real types
4.05.07	(07)	BI	WI	0021/12	88-11-04	MACHINE_OVERFLOWS for correct extended safe results
4.05.07	(07)	NB	WJ	0387/05	87-02-23	Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR
4.05.07	(08)	ra	RE	0432/00	86-06-19	A SAFE interval gives the bounds required on the result?
4.05.07	(09)	ra	CA	0476/03	88-05-02	Model interval for exponentiation with a negative exponent
4.05.07	(10)	BI	CA	0174/07	88-10-03	T'FIRST and T'LAST for real types
4.05.07	(12)	ra	RE	0525/00	87-02-10	Unnecessary Loss of Accuracy

4.6 Type Conversions

The evaluation of an explicit type conversion evaluates the expression given as the operand, and converts the resulting value to a specified *target* type. Explicit type conversions are allowed between closely related types as defined below.

type_conversion ::= type_mark(expression)

The target type of a type conversion is the base type of the type mark. The type of the operand of a type conversion must be determinable independently of the context (in particular, independently of the target type). Furthermore, the operand of a type conversion is not allowed to be ♦ an aggregate or a string literal; an expression enclosed by parentheses is allowed as the operand of a type conversion only if the expression alone is allowed.

A conversion to a subtype consists of a conversion to the target type followed by a check that the result of the conversion belongs to the subtype. A conversion of an operand of a given type to the type itself is

allowed.

The other allowed explicit type conversions correspond to the following three cases:

(a) Numeric types

The operand can be of any numeric type; the value of the operand is converted to the target type which must also be a numeric type. ♦

(b) Omitted

(c) Array types

The conversion is allowed if the operand type and the target type are array types that satisfy the following conditions: both types must have the same dimensionality; for each index position the index types must either be the same or be convertible to each other; the component types must be the same ♦. If the type mark denotes an unconstrained array type, then, for each index position, the bounds of the result are obtained by converting the bounds of the operand to the corresponding index type of the target type. If the type mark denotes a constrained array subtype, then the bounds of the result are those imposed by the type mark. In either case, the value of each component of the result is that of the matching component of the operand (see 4.5.2).

In the case of conversions of numeric types ♦, the exception CONSTRAINT_ERROR is raised by the evaluation of a type conversion if the result of the conversion fails to satisfy a constraint imposed by the type mark.

In the case of array types, a check is made that any constraint on the component subtype is the same for the operand array type as for the target array type. If the type mark denotes an unconstrained array type and if the operand is not a null array, then, for each index position, a check is made that the bounds of the result belong to the corresponding index subtype of the target type. If the type mark denotes a constrained array subtype, a check is made that for each component of the operand there is a matching component of the target subtype, and vice versa. The exception CONSTRAINT_ERROR is raised if any of these checks fails.

If a conversion is allowed from one type to another, the reverse conversion is also allowed. ♦

Apart from the explicit type conversions and implicit subtype conversions in the case of assignment with array types (see 5.2.1), the only allowed form of type conversion is the implicit conversion of a value of the type *universal_integer* ♦ into another numeric type. An implicit conversion of an operand of type *universal_integer* to another integer type ♦ can only be applied if the operand is either a numeric literal, a named number, ♦ an attribute or the application of an implementation defined attribute that is a function returning a value of type *universal_integer* [AI-00218]; such an operand is called a *convertible* universal operand in this section. An implicit conversion of a convertible universal operand is applied if and only if the innermost complete context (see 8.7) determines a unique (numeric) target type for the implicit conversion, and there is no legal interpretation of this context without this conversion.

Notes:

The rules for implicit conversions imply that no implicit conversion is ever applied to the operand of an explicit type conversion. Similarly, implicit conversions are not applied if both operands of a predefined relational operator are convertible universal operands.

The language allows implicit subtype conversions in the case of array types (see 5.2.1). ♦ Explicit conversions are NOT used for actual parameters that correspond to formals of mode **in out** (see 6.4).

♦ 18

Examples of conversions between array types: 20

```
type SEQUENCE is array (INTEGER range <>) of INTEGER;
subtype DOZEN is SEQUENCE(1 .. 12);
subtype ledger_index is integer range (1..100);
LEDGER : array(ledger_index) of INTEGER;

SEQUENCE(LEDGER)          --  bounds are those of LEDGER
```
♦

Examples of implicit conversions: 21

```
X : INTEGER := 2;

X + 1 + 2                 -- implicit conversion of each universal_integer literal
1 + 2 + X                 -- implicit conversion of each universal_integer literal
X + (1 + 2)               -- implicit conversion of each universal_integer literal

2 = (1 + 1)               -- no implicit conversion:  the type is universal_integer
A'LENGTH = B'LENGTH       -- no implicit conversion:  the type is universal_integer
C : constant := 3 + 2;    -- no implicit conversion:  the type is universal_integer

X = 3 and 1 = 2           -- implicit conversion of 3, but not of 1 and 2
```

References: actual parameter 6.4.1, array type 3.6, attribute 4.1.4, base type 3.3, belong to a subtype 3.3, component 3.3, constrained array subtype 3.6, constraint_error exception 11.1, dimension 3.6, expression 4.4, index 3.6, index subtype 3.6, index type 3.6, integer type 3.5.4, matching component 4.5.2, mode 6.1, name 4.1, named number 3.2, null array 3.6.1, numeric literal 2.4, numeric type 3.5, raising of exceptions 11, statement 5, subtype 3.3, type 3.3, type mark 3.3.2, unconstrained array type 3.6, universal_integer type 3.5.4, variable 3.2.1 22

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.06	(03)	ra	CA	0481/04	88-10-03	Overload resolution for the operand of a type conversion
4.06	(07)	BI	RE	0601/00	88-11-08	Accuracy of type conversions from real types to integers
4.06	(07)	ST	RE	0526/00	87-03-11	Rounding up or down
4.06	(11)	na	na	0368/07	87-03-06	NUMERIC_ERROR for array and integer conversions
4.06	(13)	BI	WJ	0313/03	86-07-23	Non-null bounds belong to the index subtype
4.06	(13)	co	WA	0022/05	88-11-04	Checking the component subtypes in array conversions
4.06	(15)	BI	RE	0569/00	88-07-06	Rules of the form "X must be of type T"
4.06	(15)	BI	RE	0606/00	88-12-13	Implicit conversion rules
4.06	(15)	BI	WA	0218/08	88-11-08	Implicit conversion of attributes that are functions
4.06	(15)	co	WA	0165/04	88-11-04	Expressions and implicit conversions
4.06	(15)	na	na	0250/01	86-03-25	[combined with 0218]
4.06	(15)	ra	WA	0136/05	88-11-04	Implicit conversion rules

4.7 Qualified Expressions

A qualified expression is used to state explicitly the type, and possibly the subtype, of an operand that is the given expression or aggregate. 1

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

The operand must have the same type as the base type of the type mark. The value of a qualified expression is the value of the operand. The evaluation of a qualified expression evaluates the operand and checks that its value belongs to the subtype denoted by the type mark. The exception CONSTRAINT_ERROR is raised if this check fails. 3

Examples: 4

```
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
```
♦

Notes:

Whenever the type of an enumeration literal or aggregate is not known from the context, a qualified expression can be used to state the type explicitly. For example, an overloaded enumeration literal must be qualified in the following cases: when given as a parameter in a subprogram call to an overloaded subprogram that cannot otherwise be identified on the basis of remaining parameter or result types, in a relational expression where both operands are overloaded enumeration literals, or in an array or loop parameter range where both bounds are overloaded enumeration literals. Explicit qualification is also used ♦ to constrain a value to a given subtype. 5

References: aggregate 4.3, array 3.6, base type 3.3, bound of a range 3.5, constraint_error exception 11.1, context of overload resolution 8.7, enumeration literal 3.5.1, expression 4.4, function 6.5, loop parameter 5.5, overloading 8.5, raising of exceptions 11, range 3.3, relation 4.4, subprogram 6, subprogram call 6.4, subtype 3.3, type 3.3, type mark 3.3.2 6

4.8 Allocators: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.08	(05)	BI	WJ	0150/04	86-07-23	Allocated objects belong to the designated subtype
4.08	(05)	BI	WJ	0397/04	86-07-23	Checking the designated subtype for an allocator
4.08	(05)	ra	WJ	0331/07	87-06-18	The effect of a constraint in an allocator
4.08	(06)	BI	RE	0580/00	88-08-31	Undefined variables created by allocators
4.08	(07)	BI	WJ	0356/08	88-05-23	Access values that designate deallocated objects
4.08	(07)	ST	RE	0570/00	88-07-06	Releasing heap storage associated with task type instances
4.08	(11)	ra	WJ	0294/05	87-08-20	The name given in pragma CONTROLLED
4.08	(13)	BI	WJ	0397/04	86-07-23	Checking the designated subtype for an allocator

4.9 Static Expressions and Static Subtypes

Certain expressions of a scalar type are said to be *static*. Similarly, certain discrete ranges are said to be static, and the type marks of certain scalar subtypes are said to denote static subtypes. 1

An expression of a scalar type is said to be static if and only if every primary is one of those listed in (a) through (h) below, every operator denotes a predefined operator, the expression contains no membership test (**in**, **not in**) or short-circuit control forms [AI-00128], the evaluation of the expression delivers a value (that is, it does not raise an exception), and every component factor, term, simple expression and relation has a scalar type [AI-00219]: 2

(a) An enumeration literal (including a character literal). 3

(b) A numeric literal. 4

(c) A named number. 5

(d) A name denoting a constant explicitly declared by a constant declaration with a static subtype, and initialized with a static expression [AI-00001]. 6

(e) A function call whose function name is an operator symbol that denotes a predefined operator, including a function name that is an expanded name or a name that denotes an enumeration literal, a predefined operator, or a language defined attribute of a static subtype [AI-00438, AI-00492]; each actual parameter must also be a static expression. 7

(f) A language-defined attribute of a static subtype; for an attribute that is a function, the actual parameter must also be a static expression. 8

(g) A qualified expression whose type mark denotes a static subtype and whose operand is a static expression. 9

(h) A static expression enclosed in parentheses. 10

A static range is a range whose bounds are static expressions. A static range constraint is a range constraint whose range is static. A static subtype is either a scalar base type ♦; or a scalar subtype formed by imposing on a static subtype ♦ a static range constraint. A static discrete range is either a static subtype or a static range. A static index constraint is an index constraint for which each index subtype of the corresponding array type is static, and in which each discrete range is static. If the range constraint in a subtype indication is not compatible with the type mark, the subtype defined by the subtype indication is not static [AI-00114]. 11

Notes:

♦ 12

Array attributes are not static: in particular, the RANGE attribute is not static. 13

References: actual parameter 6.4.1, attribute 4.1.4, base type 3.3, bound of a range 3.5, character literal 2.5, constant 3.2.1, constant declaration 3.2.1, discrete range 3.6, discrete type 3.5, enumeration literal 3.5.1, exception 11, expression 4.4, function 6.5, implicit declaration 3.1, initialize 3.2.1,, named number 3.2, numeric literal 2.4, predefined operator 4.5, qualified expression 4.7, raising of exceptions 11, range constraint 3.5, scalar type 3.5, subtype 3.3, type mark 3.3.2 14

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.09	(02)	BI	WJ	0128/04	86-07-23	No membership tests or short-circuit operations in static expressions
4.09	(02)	BI	WJ	0190/05	86-07-23	A static expression cannot have a generic formal type

4.09	(02)	BI	WJ	0219/06	86-07-23	Use of & and 'IMAGE in static expressions
4.09	(02)	co	RE	0607/00	88-12-13	Is a renamed predefined operator predefined?
4.09	(06)	BI	WA	0505/03	88-06-16	Static constants in instances
4.09	(06)	BI	WJ	0001/10	86-07-23	Renaming and static expressions
4.09	(06)	na	na	0208/01	86-08-21	[combined with 0163]
4.09	(06)	ra	WJ	0163/05	86-07-23	Implicit conversion preserves staticness
4.09	(07)	BI	CA	0438/08	88-10-17	Static function calls
4.09	(07)	BI	WA	0492/04	88-06-16	PRED, SUCC, POS, and VAL can be used in static expressions
4.09	(11)	BI	CA	0114/05	88-10-03	A subtype indication with an incompatible range is not static
4.09	(11)	BI	WJ	0023/06	86-07-23	Static numeric subtypes
4.09	(11)	BI	WJ	0251/05	87-03-16	Are types derived from generic formal types static subtypes?
4.09	(11)	BI	WJ	0409/05	87-09-12	Static subtype names created by instantiation
4.09	(11)	ST	RE	0539/00	87-08-05	Need for static attributes of arrays and records

4.10 Universal Expressions

A *universal_expression* is an expression that delivers a result of type *universal_integer* ♦. 1

The same operations are predefined for the type *universal_integer* as for any integer type. ♦ 2

For the evaluation of an operation of a ♦ universal expression, an implementation is required[30] to raise the exception CONSTRAINT_ERROR[31] ♦ if the result or any operand [AI-00181] of the operation is an integer value greater than AVA.MAX_INT or less than AVA.MIN_INT.[32] 5

♦ 6

Examples: 7

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

KILO : constant := 1000;
MEGA : constant := KILO*KILO;  -- 1_000_000
♦
```

References: actual parameter 6.4.1, attribute 4.1.4, ava.max_int 13.7, ava.min_int 13.7, evaluation of an expression 4.5, function 6.5, integer type 3.5.4, multiplying operator 4.5 4.5.5, predefined operation 3.3.3, primary 4.4, type 3.3, universal_integer type 3.5.4 8

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
4.10	(02)	ra	WA	0129/04	88-11-04	2.0**3 can be a *universal_real* expression
4.10	(04)	BI	CA	0565/02	88-10-03	Support for static *universal_integer* expressions
4.10	(04)	ra	WJ	0103/06	86-07-23	Accuracy of a relation between two static universal real operands
4.10	(04)	ra	WJ	0209/06	88-05-23	Exact evaluation of static universal real expressions
4.10	(04)	ra	WJ	0405/06	86-12-01	One nonstatic operand for a universal real relation
4.10	(04)	ST	RE	0285/00	84-10-01	Accuracy of Attributes of Generic Formal Types

---

[30]**IMPLEMENTATION REQUIREMENT**: Ada permits static universal expression to deliver results outside the bounds of "the largest value of all predefined integer types". AVA does not allow this for *either* static or non-static expressions.

[31]AI-00387.

[32]Note that the Ada definition is ambiguous as to whether certain expressions are static or not. Consider "1E100 - 1E100". It can be considered either 1) static, with value 0. or 2) non-static, raising CONSTRAINT_ERROR. The choice is implementation dependent.

4.10	(05)	BI	WJ	0181/04	86-07-23	NUMERIC_ERROR for nonstatic universal operands
4.10	(05)	NB	WJ	0387/05	87-02-23	Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR

Chapter 5
STATEMENTS

A *statement* defines an action to be performed; the process by which a statement achieves its action is called *execution* of the statement. 1

This chapter describes the general rules applicable to all statements. Some specific statements are discussed in later chapters. Procedure call statements are described in Chapter 6 on subprograms. ♦ Raise statements are described in Chapter 11 on exceptions ♦. The remaining forms of statements are presented in this chapter. 2

References: ♦ procedure call statement 6.4, raise statement 11.3 3

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

sequence_of_statements ::= statement {statement} 2

statement ::=
♦ simple_statement | ♦ compound_statement

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

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

♦

null_statement ::= **null**;

♦ For a block statement without a declarative part, an implicit declarative part (and preceding **declare**) is assumed. 3

♦ 4

Execution of a null statement has no other effect than to pass to the next action. 5

The execution of a sequence of statements consists of the execution of the individual statements in succession until the sequence is completed, or a transfer of control takes place. A transfer of control is caused either by the execution of an exit ♦ or return statement; ♦ or by the raising of an exception ♦. 6

♦ 7

Note:

The scope of a declaration starts at the place of the declaration itself (see 8.2). ♦ An implicit declaration in a block statement may hide a declaration given in an outer program unit or block statement (according to the usual rules of hiding explained in section 8.3). 8

References: assignment statement 5.2, ♦ block statement 5.6, case statement 5.4, declaration 3.1, declarative part 3.9, exception 11, exit statement 5.7, hiding 8.3, identifier 2.3, if statement 5.3, implicit declaration 3.1, ♦ loop statement 5.5, package 7, package body 7.1, procedure call statement 6.4, program unit 6, raise statement 11.3, raising of exceptions 11, return statement 5.8, scope 8.2, simple name 4.1, subprogram 6, subprogram body 6.3 9

5.2 Assignment Statement

An assignment statement replaces the current value of a variable with a new value specified by an expression. The named variable and the right-hand side expression must be of the same type ♦.[33] 1

```
assignment_statement ::=                                    2
  variable_name := expression;
```

For the execution of an assignment statement, the variable name and the expression are first evaluated, in that order.[34] It is poor programming style if the evaluation of variable name and the expression modify variables that are mutually accessible. A check is then made that the value of the expression belongs to the subtype of the variable, except in the case of a variable that is an array (the assignment then involves a subtype conversion as described in section 5.2.1). Finally, the value of the expression becomes the new value of the variable. 3

The exception CONSTRAINT_ERROR is raised if the above-mentioned subtype check fails; in such a case the current value of the variable is left unchanged. ♦ 4

Examples: 5

```
VALUE := MAX_VALUE - 1;
SHADE := BLUE;
♦
U := DOT_PRODUCT(V, W);          --  see 6.5
♦
BIRTHDATE := (DAY => 1, MONTH => MAY, YEAR => 1960);  -- see 3.7
```

Examples of constraint checks: 6

---

[33]Note that in full Ada the type of the expression can be used to determine an overload resolution of the left hand side in certain circumstances [AI-00120]. But this case does not arise in AVA.

[34]**IMPLEMENTATION REQUIREMENT**. Note that ‘‘evaluate’’ is not an accurate description of what we do to the variable name in order to find out its type constraints.

```
I, J    : INTEGER range 1 .. 10 := 1;
K       : INTEGER range 1 .. 20 := 11;

...

I := J;       -- identical ranges
K := J;       -- compatible ranges
J := K;       -- will raise the exception CONSTRAINT_ERROR if K > 10
```

Notes:

♦ 7

If the right-hand side expression is either a numeric literal or named number, or an attribute that yields a result of type *universal_integer* ♦, then an implicit type conversion is performed, as described in section 4.6. 8

♦ 9

References: array 3.6, array assignment 5.2.1, component 3.6 3.7, constraint_error exception 11.1, designate 3.8, evaluation 4.5, expression 4.4, function call 6.4, implicit type conversion 4.6, name 4.1, numeric literal 2.4, object 3.2, overloading 6.6 8.7, subcomponent 3.3, subtype 3.3, subtype conversion 4.6, type 3.3, universal_integer type 3.5.4, variable 3.2.1 10

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
5.02	(03)	BI	WA	0333/08	88-11-04	Evaluation order when assigning an array aggregate
5.02	(03)	BI	WI	0571/02	88-12-13	Floating point subtypes with reduced accuracy
5.02	(03)	BI	WJ	0407/06	88-05-23	The operations of a subtype with reduced accuracy
5.02	(04)	BI	RE	0586/00	88-09-02	meaning of ’these discriminants’
5.02	(04)	ra	RE	0587/00	88-09-02	meaning of ’this execution’

5.2.1 Array Assignments

If the variable of an assignment statement is an array variable ♦, the value of the expression is implicitly converted to the subtype of the array variable; the result of this subtype conversion becomes the new value of the array variable. 1

This means that the new value of each component of the array variable is specified by the matching component in the array value obtained by evaluation of the expression (see 4.5.2 for the definition of matching components). The subtype conversion checks that for each component of the array variable there is a matching component in the array value, and vice versa. The exception CONSTRAINT_ERROR is raised if this check fails; in such a case the value of each component of the array variable is left unchanged. 2

Examples: 3

```
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.     ";
```

```
B : S2 := A;                    --  same number of components
```

Notes:

♦ 4

The implicit subtype conversion described above for assignment to an array variable is performed only for the value of the right-hand side expression as a whole; it is not performed for subcomponents that are array values. Such subcomponents must be of the same subtype in the expression and variable. 5

References: array 3.6, assignment 5.2, constraint_error exception 11.1, matching array components 4.5.2, subtype conversion 4.6, type 3.3, variable 3.2.1 6

5.3 If Statements

An if statement selects for execution one or none of the enclosed sequences of statements, depending on the (truth) value of one or more corresponding conditions. 1

```
if_statement ::=                                                    2
   if condition then
     sequence_of_statements
   {elsif condition then
     sequence_of_statements}
   [else
     sequence_of_statements]
   end if;

condition ::= boolean_expression
```

An expression specifying a condition must be of a boolean type.[35] 3

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 one condition evaluates to TRUE, then the corresponding sequence of statements is executed; otherwise none of the sequences of statements is executed. 4

Examples: 5

```
if MONTH = DECEMBER and DAY = 31 then
   MONTH := JANUARY;
   DAY   := 1;
   YEAR  := YEAR + 1;
end if;

if LINE_TOO_SHORT then
   raise PROGRAM_ERROR;
elsif LINE_FULL then
    PUT(OFILE,EOL);
    PUT(OFILE,ITEM);
```

[35] Note that in full Ada different conditions do not need to be of the same boolean type.

```
else
  PUT(OFILE,ITEM);
end if;

if NEXT_PERSON.VEHICLE.OWNER /= NEXT_PERSON.NAME then     -- see 3.7
  REPORT ("Incorrect data");
end if;
```

References: boolean type 3.5.3, evaluation 4.5, expression 4.4, sequence of statements 5.1 6

5.4 Case Statements

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. 1

```
case_statement ::=                                                    2
  case expression is
     case_statement_alternative
    {case_statement_alternative}
  end case;

case_statement_alternative ::=
  when choice {| choice } =>
    sequence_of_statements
```

The expression must be of a discrete type which must be determinable independently of the context in which the expression occurs, but using the fact that the expression must be of a discrete type. ♦ Each choice in a case statement alternative must be of the same type as the expression (unless it is a discrete range, in which case the elements of the range must be of the same type as the expression); the list of choices specifies for which values of the expression the alternative is chosen. 3

If the expression is the name of an object whose subtype is static, then each value of this subtype must be represented once and only once in the set of choices of the case statement, and no other value is allowed; this rule is likewise applied if the expression is a qualified expression or type conversion whose type mark denotes a static subtype. Otherwise, for other forms of expression, each value of the (base) type of the expression must be represented once and only once in the set of choices, and no other value is allowed. 4

The simple expressions and discrete ranges given as choices in a case statement must be static. A choice defined by a discrete range stands for all values in the corresponding range (none if a null range). The choice **others** is only allowed for the last alternative and as its only choice; it stands for all values (possibly none) not given in the choices of previous alternatives.[36] A component simple name is not allowed as a choice of a case statement alternative. 5

The execution of a case statement consists of the evaluation of the expression followed by the execution of the chosen sequence of statements. 6

Examples: 7

```
case SENSOR is
```

[36]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.

```
   when ELEVATION  => RECORD_ELEVATION(SENSOR_VALUE);
   when AZIMUTH    => RECORD_AZIMUTH  (SENSOR_VALUE);
   when DISTANCE   => RECORD_DISTANCE (SENSOR_VALUE);
   when others     => null;
end case;

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;

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;
```

Notes:

The execution of a case statement chooses one and only one alternative, since the choices are exhaustive and mutually exclusive. 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. 8

An **others** choice is required in a case statement if the type of the expression is the type *universal_integer* (for example, if the expression is an integer literal), since this is the only way to cover all values of the type universal_integer. 9

References: base type 3.3, choice 4.3, context of overload resolution 8.7, discrete type 3.5, expression 4.4, function call 6.4, conversion 4.6, discrete type 3.5, enumeration literal 3.5.1, expression 4.4, name 4.1, object 3.2.1, overloading 6.6 8.7, qualified expression 4.7, sequence of statements 5.1, static discrete range 4.9, static subtype 4.9, subtype 3.3, type 3.3, type conversion 4.6, type mark 3.3.2 10

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
5.04	(03)	BI	WJ	0151/05	86-07-23	Case expression of a type derived from a generic formal type
5.04	(05)	ST	RE	0477/00	86-10-13	Case choises should not have to be static.
5.04	(06)	ra	WJ	0267/06	88-07-07	Evaluating expressions in case statements

5.5 Loop Statements

A loop statement includes a sequence of statements that is to be executed repeatedly, zero or more times. 1

```
loop_statement ::=                                                    2
♦  [iteration_scheme] loop
      sequence_of_statements
   end loop ♦;

iteration_scheme ::= while condition
  | for loop_parameter_specification

loop_parameter_specification ::=
  identifier in [reverse] discrete_range
```

♦ 3

A loop statement without an iteration scheme specifies repeated execution of the sequence of statements. Execution of the loop statement is complete when the loop is left as a consequence of the execution of an exit statement, or as a consequence of some other transfer of control (see 5.1). 4

For 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. 5

For a loop statement with a **for** iteration scheme, the loop parameter specification is the declaration of the *loop parameter* with the given identifier. The loop parameter is an object whose subtype is the discrete range [AI-00006], and thus whose type is the base type of the discrete range (see 3.6.1). Within the sequence of statements, the loop parameter is a constant. Hence a loop parameter is not allowed as the (left-hand side) variable of an assignment statement. Similarly the loop parameter must not be given as an ♦ **in out** parameter of a procedure ♦. 6

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 evaluates the discrete range. 7

If the discrete range is 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 range (subject to the loop not being left as a consequence of the execution of an exit statement or as a consequence of some other transfer of control). Prior to each such iteration, the corresponding value of the discrete range 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. 8

Example of a loop statement without an iteration scheme: 9

```
loop
  GET(STANDARD_INPUT,CURRENT_CHARACTER);
  exit when CURRENT_CHARACTER = '*';
end loop;
```

Example of a loop statement with a while iteration scheme: 10

```
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: 11

```
for J in BUFFER'RANGE loop    --  legal even with a null range
  if BUFFER(J) /= SPACE then
    PUT(STANDARD_OUTPUT,BUFFER(J));
  end if;
end loop;
```

♦ 12

Notes:

The scope of a loop parameter extends from the loop parameter specification to the end of the loop 13

statement, and the visibility rules are such that a loop parameter is only visible within the sequence of statements of the loop.

The discrete range of a for loop is evaluated just once. Use of the reserved word **reverse** does not alter the discrete range, so that the following iteration schemes are not equivalent; the first has a null range. 14

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

♦ 15

References: actual parameter 6.4.1, assignment statement 5.2, base type 3.3, bound of a range 3.5, condition 5.3, constant 3.2.1, context of overload resolution 8.7, conversion 4.6, declaration 3.1, discrete range 3.6.1, elaboration 3.1, evaluation 4.5, exit statement 5.7, expanded name 4.1.3, false boolean value 3.5.3, identifier 2.3, integer type 3.5.4, null range 3.5, object 3.2.1, prefix 4.1, procedure call 6.4, raising of exceptions 11, reserved word 2.9, return statement 5.8, scope 8.2, sequence of statements 5.1, ♦ terminate alternative 9.7.1, true boolean value 3.5.3 3.5.4, visibility 8.3 16

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
5.05	(06)	BI	WJ	0006/05	86-07-23	The subtype of a loop parameter
5.05	(06)	ra	CE	0369/06	88-11-20	Representing values of discrete base type

5.6 Block Statements

A block statement encloses a sequence of statements optionally preceded by a declarative part and optionally followed by exception handlers. 1

```
block_statement ::= ♦
    [declare
        inner_declarative_part]
     begin
        sequence_of_statements
    [exception
        exception_handler ♦]
     end ♦;
```

2

♦ 3

The execution of a block statement consists of the elaboration of its declarative part (if any) followed by the execution of the sequence of statements. If the block statement has exception handlers, these service corresponding exceptions that are raised during the execution of the sequence of statements (see 11.2). 4

Example: 5

```
♦
 declare
   TEMP : INTEGER := 1;
 begin
   TEMP := V; V := U; U := TEMP;
 end ;  ♦
```

♦ 6

References: declarative part 3.9, exception handler 11.2, exit statement 5.7, expanded name 4.1.3, raising of exceptions 11, return statement 5.8, sequence of statements 5.1, simple name 4.1 8

5.7 Exit Statements

An exit statement is used to complete the execution of an enclosing loop statement (called the loop in what follows); the completion is conditional if the exit statement includes a condition. 1

```
exit_statement ::=                                                    2
  exit ♦ [when condition];
```

♦ An exit statement ♦ is only allowed within a loop, and applies to the innermost enclosing loop. ♦ 3

For the execution of an exit statement, the condition, if present, is first evaluated. Exit from the loop then takes place if the value is TRUE or if there is no condition. 4

Examples: 5

```
for N in 1 .. MAX_NUM_ITEMS loop
  GET_NEW_ITEM(NEW_ITEM);
  MERGE_ITEM(NEW_ITEM, STORAGE_FILE);
  exit when NEW_ITEM = TERMINAL_ITEM;
end loop;

♦
loop
    -- initial statements
    exit ♦ when FOUND;
    -- final statements
  end loop ♦;
```

♦ 6

References: condition 5.3, evaluation 4.5, ♦ loop statement 5.5, package body 7.1, subprogram body 6.3, true boolean value 3.5.3 7

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
5.07	(00)	ST	RE	0211/00	84-03-13	Additional control statement for use w/in a LOOP statement.
5.07	(02)	co	WJ	0210/04	86-12-01	Loop name in an exit statement as an expanded name

5.8 Return Statements

A return statement is used to complete the execution of the innermost[37] enclosing function or procedure ♦. 1

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

A return statement is only allowed within the body of a subprogram ♦ and applies to that body. 3

[37] In AVA there can only be a *single* enclosing function or procedure.

A return statement for ♦ the body of a procedure ♦ must not include an expression. A return statement for the body of a function ♦ must include an expression. 4

The value of the expression defines the result returned by the function. The type of this expression must be the base type of the type mark given after the reserved word **return** in the specification of the function ♦ (this type mark defines the result subtype). 5

For the execution of a return statement, the expression (if any) is first evaluated and a check is made that the value belongs to the result subtype. The execution of the return statement is thereby completed if the check succeeds; so also is the execution of the subprogram ♦. The exception CONSTRAINT_ERROR is raised at the place of the return statement if the check fails. 6

Examples: 7

```
return;                                -- in a procedure
return KEY_VALUE(LAST_INDEX);          -- in a function
```

Note:

If the expression is either a numeric literal or named number, or an attribute that yields a result of type *universal_integer* ♦, then an implicit conversion of the result is performed as described in section 4.6. 8

References: block statement 5.6, constraint_error exception 11.1, expression 4.4, function body 6.3, function call 6.4, implicit type conversion 4.6, named number 3.2, numeric literal 2.4, ♦ procedure body 6.3, reserved word 2.9, result subtype 6.1, subprogram body 6.3, subprogram specification 6.1, subtype 3.3, type mark 3.3.2, universal_integer type 3.5.4 9

5.9 Goto Statements: Removed

Chapter 6
SUBPROGRAMS

Subprograms are one of the ♦ forms of *program unit*, of which programs can be composed. The other form is packages. 1

A subprogram is a program unit 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 calling conventions, and a subprogram body defining its execution. 2

References: function 6.5, function call 6.4, package 7, procedure 6.1, procedure call 6.4, subprogram body 6.3, subprogram call 6.4, subprogram declaration 6.1 3

6.1 Subprogram Declarations

A subprogram declaration declares a procedure or a function, as indicated by the initial reserved word. 1

subprogram_declaration ::= subprogram_specification; 2

subprogram_specification ::=
  procedure identifier [formal_part]
| **function** identifier [formal_part] **return** type_mark

♦

formal_part ::=
  (parameter_specification {; parameter_specification})

parameter_specification ::=
  identifier_list : mode type_mark ♦

mode ::= [**in**] | **in out** | ♦

The specification of a procedure specifies its identifier and its *formal parameters* (if any). The specification of a function specifies its identifier, its formal parameters, of which there must be at least one[38] and the subtype of the returned value (the *result subtype*). ♦ 3

[38]This is a somewhat arbitrary simplification reflecting a bias against the use of global variables. Without globals, parameterless functions are just constants.

A parameter specification with several identifiers is equivalent to a sequence of single parameter specifications, as explained in section 3.2. Each single parameter specification declares a formal parameter. If no mode is explicitly given, the mode **in** is assumed. ♦ 4

The elaboration of a subprogram declaration elaborates the corresponding formal part. The elaboration of a formal part has no other effect. 6

Examples of subprogram declarations: 7

```
procedure TRAVERSE_TREE;
procedure INCREMENT(X : in out INTEGER);
♦
procedure SWITCH(FROM, TO : in out COLOR);                -- see 3.5.1

function RANDOM (I : in PAGE_NUM) return PAGE_NUM;        -- see 3.5.4

function BIRTH_DATE(K : PERSON) return DATE;              -- see 3.7
♦
```

Notes:

♦ 8

All subprograms can be called recursively and are reentrant. 9

References: declaration 3.1, elaboration 3.9, evaluation 4.5, expression 4.4, formal parameter 6.2, function 6.5, identifier 2.3, identifier list 3.2, mode 6.2, name 4.1, elaboration has no other effect 3.9, operator 4.5, overloading 6.6 8.7, procedure 6, string literal 2.6, subprogram call 6.4, type mark 3.3.2 10

6.2 Formal Parameter Modes

The value of an object is said to be *read* when this value is evaluated; it is also said to be read when one of its subcomponents is read. The value of a variable is said to be *updated* when an assignment is performed to the variable, and also (indirectly) when the variable is used as actual parameter of a subprogram call ♦ that updates its value; it is also said to be updated when one of its subcomponents is updated. 1

A formal parameter of a subprogram has one of the ♦ following modes: 2

in The formal parameter is a constant and permits only reading of the value of the associated actual parameter. 3

in out The formal parameter is a variable (or variable subcomponent) and permits both reading and updating of the value of the associated actual parameter.[39] 4

♦

For a scalar parameter, the above effects are achieved by copy: at the start of each call the value of the actual parameter is copied into the associated formal parameter; then after normal completion of the subprogram body, if the mode is **in out** ♦, the value of the formal parameter is copied back into the associated actual parameter. ♦ 6

[39]Note that (x) a variable if x is a variable, according to [AI-00178].

For a parameter whose type is an array ♦ or record type, an implementation must achieve the above effects by copy, as for scalar types.[40] ♦ 7

For a parameter whose type is a private type, the above effects are achieved according to the rule that applies to the corresponding full type declaration. 8

Within the body of a subprogram, a formal parameter is subject to any constraint resulting from the type mark given in its parameter specification. For a formal parameter of an unconstrained array type, the bounds are obtained from the actual parameter, and the formal parameter is constrained by these bounds (see 3.6.1). ♦ 9

Notes:

For parameters of array and record types, the parameter passing rules have these consequences: 11

- If the execution of a subprogram is abandoned as a result of an exception, the final value of an actual parameter of such a type must be its value before the call. 12
- ♦ 13

References: actual parameter 6.4.1, array type 3.6, assignment 5.2, bound of an array 3.6.1, constraint 3.3, evaluation 4.5, exception 11, expression 4.4, formal parameter 6.1, global 8.1, mode 6.1, object 3.2, parameter specification 6.1, private type 7.4, record type 3.7, scalar type 3.5, subcomponent 3.3, subprogram body 6.3, subprogram call statement 6.4, type mark 3.3.2, unconstrained array type 3.6, unconstrained variable 3.2.1, variable 3.2.1 16

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
6.02	(01)	BI	WI	0141/03	84-12-10	Meaning of "update" for shared actual parameters
6.02	(05)	ST	RE	0478/00	86-10-13	Referring to out-mode formal parameters to be allowed.
6.02	(06)	BI	RE	0348/00	85-06-18	Completion of the subprogram body
6.02	(06)	ST	RE	0349/00	85-06-18	Delete copy-in/copy-back for scalar and access parameters
6.02	(06)	ST	RE	0479/00	86-10-13	Access type out-variables should be null before call
6.02	(07)	BI	RE	0588/00	88-09-02	Dependence from parameter passing mechanism
6.02	(07)	BI	WI	0178/03	88-08-16	Passing parameters by reference
6.02	(09)	ra	WA	0005/08	88-11-04	'CONSTRAINED for a formal parameter
6.02	(13)	BI	WI	0178/03	88-08-16	Passing parameters by reference
6.02	(13)	ra	WI	0135/03	88-09-02	Erroneous use of array and record formal parameters

6.3 Subprogram Bodies

A subprogram body specifies the execution of a subprogram. 1

```
subprogram_body ::=                                                    2
   subprogram_specification is
     [inner_declarative_part]
   begin
      sequence_of_statements
   [exception
```

---

[40] **IMPLEMENTATION REQUIREMENT**. Note that if it can be demonstrated to yield *identical* results for **in out** parameters, an implementation may achieve these effects by reference. This will involve a proof of the absence of exceptions propagating out of the subprogram call. See section 1.6 for a description of a program transformation that will permit compilation and execution of AVA programs under all conforming Ada compilers.

exception_handler ♦]
end [identifier];

The declaration of a subprogram is optional. In the absence of such a declaration, the subprogram specification of the subprogram body ♦ acts like a declaration. Differences arise when discussing recompilation. See 10.1(6) and [AI-00199]. For each subprogram declaration, there must be a corresponding body ♦. If both a declaration and a body are given, the subprogram specification of the body must conform to the subprogram specification of the declaration (see section 6.3.1 for conformance rules). 3

If an identifier appears at the end of a subprogram body, it must repeat the identifier of the subprogram specification. 4

The elaboration of a subprogram body has no other effect than to establish that the body can from then on be used for the execution of calls of the subprogram. 5

The execution of a subprogram body is invoked by a subprogram call (see 6.4). For this execution, after establishing the association between formal parameters and actual parameters, the inner declarative part of the body is elaborated, and the sequence of statements of the body is then executed. Upon completion of the body, return is made to the caller (and any necessary copying back of formal to actual parameters occurs (see 6.2)). The optional exception handlers at the end of a subprogram body handle exceptions raised during the execution of the sequence of statements of the subprogram body (see 11.4). 6

Note:

It follows from the visibility rules that if a subprogram declared in a package is to be visible outside the package, a subprogram specification must be given in the visible part of the package. The same rules dictate that a subprogram declaration must be given if a call of the subprogram occurs textually before the subprogram body (the declaration must then occur earlier than the call in the program text). The rules given in sections 3.9 and 7.1 imply that a subprogram declaration and the corresponding body must both occur immediately within the same declarative region. 7

Example of subprogram body: 8

```
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;
```

References: actual parameter 6.4.1, conform 6.3.1, declaration 3.1, declarative part 3.9, declarative region 8.1, designator 6.1, elaboration 3.9, elaboration has no other effect 3.1, exception 11, exception handler 11.2, formal parameter 6.1, occur immediately within 8.1, package 7, sequence of statements 5.1, subprogram 6, subprogram call 6.4, subprogram declaration 6.1, subprogram specification 6.1, visibility 8.3, visible part 7.2 9

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
6.03	(03)	BI	RE	0252/01	84-11-24	Can a subprogram spec appear after its body

6.3.1 Conformance Rules

Whenever the language rules require or allow the specification of a given subprogram to be provided in more than one place, the following variations are allowed at each place: 1

- ♦ 3
- A simple name can be replaced by an expanded name in which this simple name is the selector, if and only if at both places the meaning of the simple name is given by the same declaration. 4
- ♦ 5

Two subprogram specifications are said to *conform* if, apart from comments and the above allowed variations, both specifications are formed by the same sequence of lexical elements, and corresponding lexical elements are given the same meaning by the visibility and overloading rules. 6

Conformance is likewise defined for formal parts ♦ and type marks (for deferred constants ♦). 5

Notes:

A simple name can be replaced by an expanded name even if the simple name is itself the prefix of a selected component. For example, Q.R can be replaced by P.Q.R if Q is declared immediately within P. 6

The following specifications do not conform since they are not formed by the same sequence of lexical elements: 7

```
procedure P(X,Y : INTEGER)
procedure P(X : INTEGER; Y : INTEGER)
procedure P(X,Y : in INTEGER)
```

Overload resolution occurs before conformance checks [AI-00183]. 7a

References: actual parameter 6.4 6.4.1, allow 1.6, comment 2.7, declaration 3.1, deferred constant 7.4.3, direct visibility 8.3, expanded name 4.1.3, formal part 6.1, lexical element 2, name 4.1, numeric literal 2.4, overloading 6.6 8.7, prefix 4.1, selected component 4.1.3, selector 4.1.3, simple name 4.1, subprogram specification 6.1, ♦ visibility 8.3 8

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
6.03.01	(03)	BI	WA	0461/07	88-11-04	When two declarations are considered the same
6.03.01	(03)	ra	CA	0241/06	88-10-03	Conformance between a subprogram declaration and its subunit
6.03.01	(04)	ra	WJ	0493/05	88-05-23	Operator symbols that represent the same operator
6.03.01	(05)	BI	CA	0547/03	88-10-05	Conformance rules and instantiated units
6.03.01	(05)	BI	WA	0461/07	88-11-04	When two declarations are considered the same
6.03.01	(05)	ra	WJ	0350/04	86-12-04	Lexical elements not changed by allowable character replacements
6.03.01	(06)	BI	WI	0318/03	88-08-17	Conformance rules and derived subprograms

6.3.2 Inline Expansion of Subprograms: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
6.03.02	(03)	BI	WJ	0200/08	86-12-01	Dependences created by inline of generic instantiations
6.03.02	(03)	BI	WJ	0242/09	87-06-18	Subprogram names allowed in pragma INLINE

6.4 Subprogram Calls

A subprogram call is either a procedure call statement or a function call; its evaluation invokes the execution of the corresponding subprogram body. The call specifies the association of the actual parameters, if any, with formal parameters of the subprogram. 1

procedure_call_statement ::= 2
*procedure*_name [actual_parameter_part];

function_call ::=
*function*_name [actual_parameter_part]

actual_parameter_part ::=
(parameter_association {, parameter_association})

parameter_association ::=
♦ actual_parameter

♦

actual_parameter ::=
expression | *variable*_name | ♦

Each parameter association associates an actual parameter with a corresponding formal parameter. A parameter association is ♦ *positional*. The actual parameter corresponds to the formal parameter with the same position in the formal part. ♦ 3

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 ♦. 5

The parameter associations of a subprogram call are evaluated left to right.[41] The language rules do not define in which order the values of **in out** ♦ parameters are copied back into the corresponding actual parameters (when this is done). 6

It is poor programming style for the evaluation of the actual parameters of the subprogram call to modify variables that are used in the evaluation of any of the other parameter expressions. Similarly, it is poor style if the actual parameters corresponding to two different formal parameters of mode **in out** overlap[42], or if an actual **in out** parameter overlaps a global 6e

[41]**IMPLEMENTATION REQUIREMENT**: This eliminates one source of incorrect order dependencies.

[42]Two variables overlap if they are equal or either is a subcomponent of the other. Thus, the array a and array element a(1) overlap. Array elements a(i) and a(j) *potentially* overlap.

accessible to the called subprogram or an actual **in** parameter that is referable[43]. Potential overlap should be detected at compile time, to the extent possible, and a warning issued.

Examples of procedure calls: 7

```
TRAVERSE_TREE;                    -- see 6.1
TABLE_MANAGER.INSERT(E);          -- see 7.5
PRINT_HEADER(128, TITLE, TRUE);   -- see 6.1
♦
```

Examples of function calls: 8

```
DOT_PRODUCT(U, V)                 -- see 6.1 and 6.5
CLOCK                             -- see 9.6
```

Note:

Due to the above anti-overlap suggestions, given 8a

```
procedure Q(x, y: in T, z: in out T);
```

the call "Q(a, a, b)" is acceptable, but "Q(a, b, b)" is poor style.

References: expression 4.4, formal parameter 6.1, formal part 6.1, name 4.1, simple name 4.1, subprogram 6, type mark 3.3.2, variable 3.2.1 9

6.4.1 Parameter Associations

Each actual parameter must have the same type as the corresponding formal parameter. 1

An actual parameter associated with a formal parameter of mode **in** must be an expression; it is evaluated before the call.[44] 2

An actual parameter associated with a formal parameter of mode **in out** ♦ must be ♦ the name of a variable ♦.[45] Type conversions of actual parameters associated with an **in out** formal parameter are not allowed. 3

The variable name given for an actual parameter of mode **in out** ♦ is ‘‘evaluated’’ before the call (see 5.2). ♦ 4

---

[43]An expression is called *referable* in the following cases:

(a) it is a variable, or

(b) it is a parenthesized referable expression, or

(c) it is a qualified expression or type conversion whose operand is referable.

[44]Evaluating or checking an actual subprogram parameter *before the call* means the evaluation or check is performed during the execution of the subprogram call, before control is transferred to the subprogram body [AI-00433]. Thus, exceptions raised during the evaluation of parameters will *not* be handled by the exception handler of the subprogram body, but by a handler that is a function of the calling environment.

[45]Note that subcomponents of variables, e.g. ‘‘a(i)’’, are themselves the names of variables.

The following constraint checks are performed for parameters of scalar ♦ type: 5

- Before the call: ♦ it is checked that the value of the actual parameter belongs to the subtype of the formal parameter. 6
- After (normal) completion of the subprogram body: for a parameter of mode **in out** ♦, it is checked that the value of the formal parameter belongs to the subtype of the actual variable before any values are copied back.[46] ♦ 7

♦ 8

For other types, for all modes, a check is made before the call as for scalar ♦ types; no check is made upon return. When a subprogram parameter has a private type, the constraint checks that are performed before or after the call are those appropriate for the type declared in the private type's full declaration [AI-00025]. 9

The exception CONSTRAINT_ERROR is raised at the place of the subprogram call if either of these checks fails. 10

Note:

For array types ♦ the check before the call is sufficient (a check upon return would be redundant) if the type mark of the formal parameter denotes a constrained subtype, since array bounds ♦ cannot then vary. 11

If this type mark denotes an unconstrained array type, the formal parameter is constrained with the bounds of the corresponding actual parameter and no check (neither before the call nor upon return) is needed (see 3.6.1). ♦ 12

AVA Note: In both Ada and AVA assignment works very differently from parameter passing. No implicit subtype conversion is performed on arrays passed as parameters. 12a

References: actual parameter 6.4, array bound 3.6, array type 3.6, call of a subprogram 6.4, conform 6.3.1, constrained subtype 3.3, constraint 3.3, constraint_error exception 11.1, evaluation 4.5, evaluation of a name 4.1, expression 4.4, formal parameter 6.1, mode 6.1, name 4.1, parameter association 6.4, subtype 3.3, type 3.3, type conversion 4.6, type mark 3.3.2, unconstrained array type 3.6, variable 3.2.1 13

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
6.04.01	(02)	co	CE	0433/03	88-11-20	Evaluating and checking parameters "before the call"
6.04.01	(03)	BI	CA	0547/03	88-10-05	Conformance rules and instantiated units
6.04.01	(03)	BI	WI	0318/03	88-08-17	Conformance rules and derived subprograms
6.04.01	(03)	ra	WJ	0245/08	88-05-23	Type conversion conformance for renamed subprogram/entry calls
6.04.01	(04)	BI	WJ	0024/09	86-07-23	Type conversions as out parameters for non-scalar types
6.04.01	(04)	BI	WJ	0295/05	88-05-23	Evaluating the variable in an actual parameter type conversion
6.04.01	(04)	co	CE	0433/03	88-11-20	Evaluating and checking parameters "before the call"
6.04.01	(04)	ra	WI	0296/02	87-01-15	Type conversions and parameters passed by reference
6.04.01	(06)	co	CE	0433/03	88-11-20	Evaluating and checking parameters "before the call"
6.04.01	(07)	ra	CE	0377/03	88-11-20	Interleaving of copy-back and constraint checks
6.04.01	(09)	BI	WJ	0025/08	87-09-12	Checking out parameter constraints for private types
6.04.01	(09)	co	CE	0433/03	88-11-20	Evaluating and checking parameters "before the call"
6.04.01	(09)	CR	WJ	0396/03	86-07-23	Correction to discussion of 0025

[46] **IMPLEMENTATION REQUIREMENT**. In Ada, copy-back and constraint checking can be interleaved in an order that is not defined by the language [AI-00377].

6.04.01	(11)	co	CE	0433/03	88-11-20	Evaluating and checking parameters "before the call"
6.04.01	(12)	co	CE	0433/03	88-11-20	Evaluating and checking parameters "before the call"

6.4.2 Default Parameters: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
6.04.02	(02)	co	CE	0433/03	88-11-20	Evaluating and checking parameters "before the call"
6.04.02	(02)	co	RE	0434/00	86-06-19	An actual parameter cannot be a value?

6.5 Function Subprograms

A function is a subprogram that returns a value (the result of the function call). The specification of a function starts with the reserved word **function**, and the parameters, if any, must have the mode **in** (whether this mode is specified explicitly or implicitly). The statements of the function body ♦ must include one or more return statements specifying the returned value. 1

The exception PROGRAM_ERROR is raised if a function body is left otherwise than by a return statement.[47] This does not apply if the execution of the function is abandoned as a result of an exception. 2

Example: 3

```
function DOT_PRODUCT(LEFT, RIGHT : TABLE) 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;
```

References: exception 11, formal parameter 6.1, function 6.1, function body 6.3, function call 6.4, function specification 6.1, mode 6.1, program_error exception 11.1, raising of exceptions 11, return statement 5.8, statement 5 4

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
6.05	(02)	BI	RE	0152/01	86-09-05	Raising PROGRAM_ERROR in a function call

[47] The exception is raised at the point of call, not within the function body. Equivalently, the error is raised when it is realized that there is no more function body to execute, which in our simple model of execution is only known outside the last handler within the body. See the unresolved discussion in [AI-00152].

6.6 Parameter and Result Type Profile - Overloading of Subprograms

Two formal parts are said to have the same *parameter type profile* if and only if they have the same number of parameters, and at each parameter position corresponding parameters have the same base type. A subprogram ♦ has the same *parameter and result type profile* as another subprogram ♦ if and only if both have the same parameter type profile, and either both are functions with the same result base type, or both are procedures. 1

The same subprogram identifier ♦ can be used in several subprogram specifications. The identifier ♦ is then said to be *overloaded*; the subprograms that have this identifier ♦ are also said to be overloaded and to overload each other. As explained in section 8.3, if two subprograms overload each other, one of them can hide the other only if both subprograms have the same parameter and result type profile (see section 8.3 for the other requirements that must be met for hiding). 2

A call to an overloaded subprogram is ambiguous (and therefore illegal) if the name of the subprogram, the number of parameter associations, the types and the order of the actual parameters, ♦ and the result type (for functions) are not sufficient to determine exactly one (overloaded) subprogram specification. 3

Examples of overloaded subprograms: 4

```
procedure PUT(X : INTEGER);
procedure PUT(X : STRING);

procedure SET(TINT   : COLOR);
procedure SET(SIGNAL : LIGHT);
```

Examples of calls: 5

```
PUT(28);
PUT("no possible ambiguity here");

♦
SET(LIGHT'(RED));
SET(COLOR'(RED));

-- SET(RED) would be ambiguous since RED may
-- denote a value either of type COLOR or of type LIGHT
```

Notes:

The notion of parameter and result type profile does not include parameter names, parameter modes, and parameter subtypes ♦. 6

Ambiguities may (but need not) arise when actual parameters of the call of an overloaded subprogram are themselves overloaded function calls, literals, or aggregates. Ambiguities may also (but need not) arise when several overloaded subprograms belonging to different packages are visible. These ambiguities can usually be resolved in several ways: qualified expressions can be used for some or all actual parameters, and for the result, if any; the name of the subprogram can be expressed more explicitly as an expanded name; finally, the subprogram can be renamed. 7

References: actual parameter 6.4.1, aggregate 4.3, base type 3.3, formal parameter 6.1, function 6.5, function call 6.4, hiding 8.3, identifier 2.3, illegal 1.6, literal 4.2, mode 6.1, overloading 8.7, package 7, parameter of a subprogram 6.2, qualified expression 4.7, renaming declaration 8.5, result subtype 6.1, 8

subprogram 6, subprogram specification 6.1, subtype 3.3, type 3.3

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
6.06	(03)	BI	WJ	0287/05	88-12-14	Resolving overloaded entry calls

6.7 Overloading of Operators: Removed

Chapter 7
PACKAGES

Packages are one of the two forms of program unit, of which programs can be composed. The other forms are subprograms ♦. 1

Packages allow the specification of groups of logically related entities. In their simplest form packages specify pools of common object and type declarations. More generally, packages can be used to specify groups of related entities including also subprograms that can be called from outside the package, while their inner workings remain concealed and protected from outside users. ♦ 2

References: ♦, program unit 6, subprogram 6 ♦, type declaration 3.3.1 3

7.1 Package Structure

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

package_declaration ::= package_specification; 2

```
package_specification ::=
   package identifier is
     {basic_declarative_item}
   [private
     {basic_declarative_item}]
   end [package_simple_name ]

package_body ::=
   package body package_simple_name is
     [declarative_part ♦ ]
   [begin
      sequence_of_statements
   [exception
      exception_handler ♦]]
   end [package_simple_name];
```

The simple name at the start of a package body must repeat the package identifier. Similarly if a simple name appears at the end of the package specification or body, it must repeat the package identifier. 3

If a subprogram declaration or a package declaration is a declarative item of a given package specification, then the body (if there is one) of the program unit declared by the declarative item must itself be a declarative item of the declarative part of the body of the given package. 4

Notes:

A simple form of package, specifying a pool of objects and types, does not require a package body. One of the possible uses of the sequence of statements of a package body is to initialize such objects. For each subprogram declaration there must be a corresponding body ♦. A body is not a basic declarative item and so cannot appear in a package specification. 5

A package declaration is either a library package (see 10.2) or a declarative item declared within another package declaration. 6

References: basic declarative item 3.9, ♦, declarative item 3.9, declarative part 3.9, exception handler 11.2, identifier 2.3, library unit 10.1, object 3.2, package body 7.3, program unit 6, proper body 3.9, sequence of statements 5.1, simple name 4.1, subprogram body 6.3, subprogram declaration 6.1, type 3.3 7

7.2 Package Specifications and Declarations

The first list of declarative items of a package specification is called the *visible part* of the package. The optional list of declarative items after the reserved word **private** is called the *private part* of the package. 1

An entity declared in the private part of a package is not visible outside the package itself (a name denoting such an entity is only possible within the package). 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). 2

The elaboration of a package declaration consists of the elaboration of its basic declarative items in the given order. 3

Notes:

The visible part of a package contains all the information that another program unit is able to know about the package. A package consisting of only a package specification (that is, without a package body) can be used to represent a group of common constants or variables, or a common pool of objects and types, as in the examples below. 4

Example of a package describing a group of common variables: 5

```
package PLOTTING_DATA is
   PEN_UP : BOOLEAN;

   CONVERSION_FACTOR,
   X_OFFSET, Y_OFFSET,
   X_MIN,    Y_MIN,
   X_MAX,    Y_MAX :  INTEGER ;     --  see  3.5.7

   subtype pindex is integer range 1..500;

   X_VALUE : array (pindex) of position;
   Y_VALUE : array (pindex) of position;

end PLOTTING_DATA;
```

Example of a package describing a common pool of objects and types: 6

```
package WORK_DATA is
        subtype DAY is integer range 1..7;
   subtype HOURS_SPENT is integer range 0 .. 24;
   type TIME_TABLE  is array (DAY) of HOURS_SPENT;
   NORMAL_HOURS  : constant TIME_TABLE := (8,8,8,8,7,0,0);
end WORK_DATA;
```

References: basic declarative item 3.9, constant 3.2.1, declarative item 3.9, direct visibility 8.3, elaboration 3.9, expanded name 4.1.3, name 4.1, number declaration 3.2.2, object declaration 3.2.1, package 7, package declaration 7.1, package identifier 7.1, package specification 7.1, scope 8.2, simple name 4.1, type declaration 3.3.1, use clause 8.4, variable 3.2.1 7

7.3 Package Bodies

In contrast to the entities declared in the visible part of a package specification, the entities declared in the package body are only visible 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 (a *package* in the usual sense), in which the logical operations available to the users are clearly isolated from the internal entities. 1

For the elaboration of a package body, its declarative part is first elaborated, and its sequence of statements (if any) is then executed. The optional exception handler ♦ at the end of a package body services exceptions raised during the execution of the sequence of statements of the package body. 2

Notes:

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. The value of such a variable remains unchanged between calls issued from outside the package to subprograms declared in the visible part. The properties of such a variable are similar to those of an "own" variable of Algol 60. 3

The elaboration of the body of a subprogram 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 3.9). 4

Example of a package: 5

```
package RATIONAL_NUMBERS is

   type RATIONAL is
     record
       NUMERATOR   : INTEGER;
       DENOMINATOR : POSITIVE;
     end record;

   function EQUAL(X,Y : RATIONAL) return BOOLEAN;

   function div   (X,Y : INTEGER)  return RATIONAL;  --  to construct a rational number

   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;

  package body RATIONAL_NUMBERS is

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

    function EQUAL(X,Y : RATIONAL) return BOOLEAN is
      U,V : RATIONAL := X;
    begin
      V := Y;
      SAME_DENOMINATOR (U,V);
      return U.NUMERATOR = V.NUMERATOR;
    end EQUAL;

    function div (X,Y : INTEGER) return RATIONAL is
    begin
      if Y > 0 then
        return (NUMERATOR => X,  DENOMINATOR => Y);
      else
        return (NUMERATOR => -X, DENOMINATOR => -Y);
      end if;
    end div;

    function plus  (X,Y : RATIONAL) return RATIONAL is ...  end;
    function minus (X,Y : RATIONAL) return RATIONAL is ...  end;
    function times (X,Y : RATIONAL) return RATIONAL is ...  end;
    function div   (X,Y : RATIONAL) return RATIONAL is ...  end;

  end RATIONAL_NUMBERS;
```

References: declaration 3.1, declarative part 3.9, elaboration 3.1 3.9, exception 11, exception handler 11.2, name 4.1, package specification 7.1, program unit 6, program_error exception 11.1, sequence of statements 5.1, subprogram 6, variable 3.2.1, visible part 7.2 6

7.4 Private Type and Deferred Constant Declarations

The declaration of a type as a private type in the visible part of a package 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). Deferred constant declarations declare constants[48] of private types. 1

```
private_type_declaration ::=                                    2
  type identifier ♦ is ♦ private;

deferred_constant_declaration ::=
```

[48]We wanted to provide deferred *objects* in order to allow variables of a private type. In the case of variables of limited privated type this is impossible (see 7.4.1(4)). Thus, the only objects of limited private type that this subset of Ada can use are constants.

identifier_list : **constant** type_mark;

A private type declaration is only allowed as a declarative item of the visible part of a package ♦. 3

The type mark of a deferred constant declaration must denote a private type ♦; a deferred constant declaration and the declaration of the corresponding private type must both be declarative items of the visible part of the same package. A deferred constant declaration with several identifiers is equivalent to a sequence of single deferred constant declarations as explained in section 3.2. 4

Examples of private type declarations: 5

type KEY **is private**;
type FILE_NAME **is** ♦ **private**;

Example of deferred constant declaration: 6

NULL_KEY : **constant** KEY;

References: constant 3.2.1, declaration 3.1, declarative item 3.9, deferred constant 7.4.3, ♦ identifier 2.3, identifier list 3.2, ♦ package 7, private type 7.4.1, program unit 6, subtype 3.3, type 3.3, type mark 3.3.2, visible part 7.2 7

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
7.04	(01)	BI	CE	0270/04	88-05-09	The type of an object declared to have a private type

7.4.1 Private Types

If a private type declaration is given in the visible part of a package, then a corresponding declaration of a type with the same identifier must appear as a declarative item of the private part of the package. The corresponding declaration must be ♦ a full type declaration ♦. In the rest of this section explanations are given in terms of full type declarations ♦. 1

A private type declaration and the corresponding full type declaration define a single type. The private type declaration, together with the visible part, define the operations that are available to outside program units (see section 7.4.2 on the operations that are available for private types). On the other hand, the full type declaration defines other operations whose direct use is only possible within the package itself. 12

♦ The declared[49] full type must not be an unconstrained array type. ♦ 13

Within the specification of the package that declares a private type and before the end of the corresponding full type declaration, a restriction applies to the use of a name that denotes the private type ♦ and, likewise, to the use of a name that denotes any type ♦ that has a subcomponent of the private type. The only allowed occurrences of such a name are in a deferred constant declaration, a type ♦ or a subprogram specification ♦; moreover, occurrences within ♦ simple expressions are not allowed. 14

The elaboration of a private type declaration creates a private type. ♦ The elaboration of the full type declaration consists of the elaboration of the type definition ♦. 15

---

[49]See [AI-00232]. The *declared* full type is distinguished from the implicitly declared type as defined in 3.3.1(4).

Notes:

It follows from the given rules that the declaration of a variable of a private type is not allowed before the full declaration of the type. ♦ 16

References: array type 3.6, conform 6.3.1, declarative item 3.9, deferred constant declaration 7.4.3, elaboration 3.9, expression 4.4, full type declaration 3.3.1, identifier 2.3, incomplete type declaration 3.8.1, ♦ name 4.1, operation 3.3, package 7, package specification 7.1, private part 7.2, private type 7.4, private type declaration 7.4, record type definition 3.7, reserved word 2.9, subcomponent 3.3, subprogram specification 6.1, subtype 3.3, subtype declaration 3.3.2, type 3.3, type declaration 3.3.1, type definition 3.3.1, unconstrained array type 3.6, variable 3.2.1, visible part 7.2 17

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
7.04.01	(01)	ST	RE	0540/00	87-08-05	The full declaration of a private type
7.04.01	(02)	BI	CE	0270/04	88-05-09	The type of an object declared to have a private type
7.04.01	(03)	co	WJ	0232/05	86-07-23	Full declarations that implicitly declare unconstrained types
7.04.01	(04)	BI	WJ	0039/12	86-07 -23	Forcing occurrences and premature uses of a type
7.04.01	(04)	BI	WJ	0153/05	86-07-23	Membership tests cannot use an incompletely declared private type
7.04.01	(04)	na	na	0246/01	84-11-05	[combined with 0039]
7.04.01	(04)	ra	WI	0517/00	87-01-20	Representation clauses for incompletely declared types
7.04.01	(04)	ra	WJ	0384/05	86-07-23	Use of an incomplete private type in a formal type declaration
7.04.01	(04)	ST	RE	0327/00	85-02-15	Instantiating with an incomplete private type
7.04.01	(04)	ST	RE	0404/00	85-12-03	Incomplete types as formal object parameters

7.4.2 Operations of a Private Type

The operations that are implicitly declared by a private type declaration include basic operations. These are the operations involved in assignment ♦, membership tests ♦, qualification, and explicit conversions. 1

For a private type T, the basic operations also include the attribute T'BASE (see 3.3.3). ♦ 2

Finally, the operations implicitly declared by a private type declaration include the predefined comparison for equality and inequality ♦. 3

The above operations, together with subprograms that have a parameter or result of the private type and that are declared in the visible part of the package, are the only operations from the package that are available outside the package for the private type. 4

Within the package that declares the private type, the additional operations implicitly declared by the full type declaration are also available. The redefinition of these implicitly declared operations is **NOT** allowed between the private type declaration and the corresponding full declaration. 5

If a composite type has subcomponents of a private type and is declared outside the package that declares the private type, then the operations that are implicitly declared by the declaration of the composite type include all operations that only depend on the characteristics that result from the private type declaration alone. (For example the operator < is not included for a one-dimensional array type.) 6

If the composite type is itself declared within the package that declares the private type (including within an inner package ♦), then additional operations that depend on the characteristics of the full type are implicitly declared, as required by the rules applicable to the composite type (for example the operator < is declared for a one-dimensional array type if the full type is discrete). These additional operations are implicitly declared at the earliest place within the immediate scope of the composite type and after the full 7

type declaration.[50]

♦ 8

Note:

A private type declaration and the corresponding full type declaration define 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* the private type with a type of a particular class (for example, as an array type) is only relevant within the package itself. 11

The consequences of this actual implementation are, however, valid everywhere. ♦ 12

Example: 13

```
package KEY_MANAGER is
   type KEY is private;
   NULL_KEY : constant KEY;
   procedure GET_KEY(K : in out KEY);
   function lt (X, Y : KEY ) return BOOLEAN;
private
   type KEY is array(1..1) of integer;
   NULL_KEY : constant KEY := key'(others => 0);
end;

package body KEY_MANAGER is
   LAST_KEY : KEY := NULL_KEY;

   procedure GET_KEY(K : in out KEY) is
   begin
      LAST_KEY(1) := LAST_KEY(1) + 1;
      K := LAST_KEY;
   end GET_KEY;

   function lt (X, Y : KEY) return BOOLEAN is
   begin
      return X(1) < Y(1);
   end lt;
end KEY_MANAGER;
```

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 ">=", or arithmetic operators. 14

♦ 15

---

[50]This ‘‘earliest declaration rule’’ is odd, but not that different from the requirement in Ada that when declaring a new integer type, the type conversion "INTEGER(x:NEW_INTEGER) return INTEGER" is declared.

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 section 7.3.) 16

Note the clumsiness of this private type. The full type definition of KEY should just be INTEGER (or NATURAL). The reason that the integer needs to be wrapped up in an array type is that AVA rules out the definition of new integer types. 16a

References: assignment 5.2, attribute 4.1.4, basic operation 3.3.3, component 3.3, composite type 3.3, conversion 4.6, declaration 3.1, declarative region 8.1, dimension 3.6, equality 4.5.2, full type 7.4.1, full type declaration 3.3.1, hiding 8.3, immediate scope 8.2, implicit declaration 3.1, incomplete type declaration 3.8.1, membership test 4.5, operation 3.3, package 7, parameter of a subprogram 6.2, predefined function 8.6, predefined operator 4.5, private type 7.4, private type declaration 7.4, program unit 6, qualification 4.7, relational operator 4.5, selected component 4.1.3, subprogram 6, visible part 7.2 17

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
7.04.02	(07)	ra	WJ	0139/04	86-07-23	The declaration of "additional operations" for access types
7.04.02	(07)	ra	WJ	0154/06	86-07-23	Additional operations for composite and access types
7.04.02	(08)	ra	WJ	0154/06	86-07-23	Additional operations for composite and access types
7.04.02	(09)	co	WJ	0026/07	87-06-18	Effect of full type decl on CONSTRAINED attribute

7.4.3 Deferred Constants

If a deferred constant declaration is given in the visible part of a package then a constant declaration (that is, an object declaration declaring a constant object, with an explicit initialization) with the same identifier must appear as a declarative item of the private part of the package. This object declaration is called the *full* declaration of the deferred constant. The type mark given in the full declaration must conform to that given in the deferred constant declaration (see 6.3.1). Multiple or single declarations are allowed for the deferred and the full declarations, provided that the equivalent single declarations conform. 1

Within the specification of the package that declares a deferred constant and before the end of the corresponding full declaration, the use of a name that denotes the deferred constant is not allowed. 2

The elaboration of a deferred constant declaration has no other effect. 3

The execution of a program raises PROGRAM_ERROR if it attempts to use the value of a deferred constant before the elaboration of the corresponding full declaration. If compilers can detect this situation at compile time they should issue a warning. 4

Note:

The full declaration for a deferred constant that has a given private type must not appear before the corresponding full type declaration. This is a consequence of the rules defining the allowed uses of a name that denotes a private type (see 7.4.1). 5

References: conform 6.3.1, constant declaration 3.2.1, declarative item 3.9, deferred constant 7.4, deferred constant declaration 7.4, elaboration has no other effect 3.1, formal parameter 6.1, identifier 2.3, object declaration 3.2.1, package 7, package specification 7.1, private part 7.2, record component 3.7, type mark 3.3.2, visible part 7.2 6

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
7.04.03	(04)	BI	WJ	0155/08	86-12-01	Evaluation of an attribute prefix having an undefined value

7.4.4 Limited Types: Removed.

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
7.04.04	(04)	BI	WJ	0260/06	86-07-23	Limited "full types"
7.04.04	(04)	CR	WA	0527/04	88-06-17	Error in 0260/06 example

7.5 Example of a Table Management Package

The following example illustrates the use of packages in providing high level procedures with a simple interface to the user. 1

The problem is to define a table management package for inserting and retrieving items. The items are inserted into the table as they are supplied. Each inserted item has an order number. The items are retrieved according to their order number, where the item with the lowest order number is retrieved first. 2

From the user's point of view, the package is quite simple. There is a type called ITEM designating table items, a procedure INSERT for inserting items, and a procedure RETRIEVE for obtaining the item with the lowest order number. There is a special item NULL_ITEM that is returned when the table is empty♦. The exception PROGRAM_ERROR is raised by INSERT if the table is already full. 3

A sketch of such a package is given below. Only the specification of the package is exposed to the user. 4

```
package TABLE_MANAGER is                                                    5

  type ITEM is
    record
      ORDER_NUM : INTEGER;
      ITEM_CODE : INTEGER;
      QUANTITY  : INTEGER;
      ITEM_TYPE : CHARACTER;
    end record;

  NULL_ITEM : constant ITEM :=
    (ORDER_NUM | ITEM_CODE | QUANTITY => 0, ITEM_TYPE => ' ');

  procedure INSERT  (NEW_ITEM   : in  ITEM);
  procedure RETRIEVE(FIRST_ITEM : in out ITEM);

  ♦
end;
```

The details of implementing such packages can be quite complex; in this case they involve a two-way linked table of internal items. A local housekeeping procedure EXCHANGE is used to move an internal item between the busy and the free lists. The initial table linkages are established by the initialization part. The package body need not be shown to the users of the package. 6

```
package body TABLE_MANAGER is                                               7
  SIZE : constant := 2000;
```

```
   subtype INDEX is INTEGER range 0 .. SIZE;

   type INTERNAL_ITEM is
     record
       CONTENT : ITEM;
       SUCC    : INDEX;
       PRED    : INDEX;
     end record;

   TABLE : array (INDEX) of INTERNAL_ITEM;
   FIRST_BUSY_ITEM : INDEX := 0;
   FIRST_FREE_ITEM : INDEX := 1;

   function FREE_LIST_EMPTY (dummy : integer) return BOOLEAN is ... end;
   function BUSY_LIST_EMPTY (dummy : integer) return BOOLEAN is ... end;
   procedure EXCHANGE (FROM : in INDEX; TO : in INDEX) is ... end;

   procedure INSERT (NEW_ITEM : in ITEM) is
   begin
     if FREE_LIST_EMPTY(0) then
       raise PROGRAM_ERROR;
     end if;
     -- remaining code for INSERT
   end INSERT;

   procedure RETRIEVE (FIRST_ITEM : in out ITEM) is ... end;

 begin
   -- initialization of the table linkages
 end TABLE_MANAGER;
```

7.6 Example of a Text Handling Package: Removed

Chapter 8
VISIBILITY RULES

The rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the text of the program are described in this chapter. The formulation of these rules uses the notion of a declarative region. 1

References: declaration 3.1, declarative region 8.1, identifier 2.3, scope 8.2, visibility 8.3 2

8.1 Declarative Region

A declarative region is a portion of the program text. A single declarative region is formed by the text of each of the following: 1

- A ♦ subprogram declaration or a package declaration ♦ together with the corresponding body, if any. ♦ 2
- ♦ 3
- A record type declaration, together with a corresponding private ♦ type declaration if any ♦. 4
- A renaming declaration that includes a formal part ♦. 5
- A block statement or a loop statement. 6

In each of the above cases, the declarative region is said to be *associated* with the corresponding declaration or statement. A declaration is said to *occur immediately within* a declarative region if this region is the innermost region that encloses the declaration, not counting the declarative region (if any) associated with the declaration itself. 7

A declaration that occurs immediately within a declarative region is said to be *local* to the region. Declarations in outer (enclosing) regions are said to be *global* to an inner (enclosed) declarative region. A local entity is one declared by a local declaration; a global entity is one declared by a global declaration. 8

Some of the above forms of declarative region include several disjoint parts (for example, other declarative items can be between the declaration of a package and its body). Each declarative region is nevertheless considered as a (logically) continuous portion of the program text. Hence if any rule defines a portion of text as the text that *extends* from some specific point of a declarative region to the end of this region, then this portion is the corresponding subset of the declarative region (for example it does not include intermediate declarative items between the two parts of a package). 9

Notes:

As defined in section 3.1, the term declaration includes basic declarations, implicit declarations, and those declarations that are part of basic declarations, for example, ♦ parameter specifications. It follows from the definition of a declarative region that a ♦ parameter specification occurs immediately within the region associated with the enclosing subprogram body ♦. 10

The package STANDARD approximates a declarative region which encloses all library units: the implicit declaration of each library unit is assumed to occur immediately within this package (see sections 8.6 and 10.1.1).[51] 11

Declarative regions can be nested within other declarative regions. For example, packages can be nested within each other, and can contain record type declarations, subprogram declarations, block statements and loop statements. 12

References: basic declaration 3.1, block statement 5.6, declaration 3.1, formal part 6.1, implicit declaration 3.1, library unit 10.1, loop statement 5.5, package 7, package body 7.1, package declaration 7.1, parameter specification 6.1, private type declaration 7.4, record type 3.7, renaming declaration 8.5, standard package 8.6, subprogram body 6.3, subprogram declaration 6.1 13

8.2 Scope of Declarations

For each form of 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 entity declared by the declaration. Furthermore, if the declaration associates some notation with a declared entity, this portion of the text is also called the scope of this notation (either an identifier, a character literal, an operator symbol, or the notation for a basic operation). Within the scope of an entity, and only there, there are places where it is legal to use the associated notation in order to refer to the declared entity. These places are defined by the rules of visibility and overloading. 1

The scope of a declaration that occurs immediately within a declarative region extends from the beginning of the declaration to the end of the declarative region; this part of the scope of a declaration is called the *immediate scope*. Furthermore, for any of the declarations listed below, the scope of the declaration extends beyond the immediate scope: 2

(a) A declaration that occurs immediately within the visible part of a package declaration. 3

(b) Omitted 4

(c) A component declaration. 5

(d) Omitted 6

(e) A parameter specification. 7

(f) Omitted 8

In each of these cases, the given declaration occurs immediately within some enclosing declaration, and the scope of the given declaration extends to the end of the scope of the enclosing declaration. 9

In the absence of a subprogram declaration, the subprogram specification given in the subprogram body ♦ acts as the declaration and rule (e) applies also in such a case. 10

---

[51] The independent compilation of library units does not allow us to use the notion ‘‘end of declarative region’’ in conjunction with the package STANDARD in any meaningful way.

Note:

The above scope rules apply to all forms of declaration defined by section 3.1; in particular, they apply also to implicit declarations. Rule (a) applies to a package declaration ♦. For nested declarations, the rules (a) through (f) apply at each level. ♦ The scope of a use clause is defined in section 8.4. 11

References: basic operation 3.3.3, character literal 2.5, component declaration 3.7, declaration 3.1, declarative region 8.1, extends 8.1, identifier 2.3, implicit declaration 3.1, occur immediately within 8.1, overloading 6.6 8.7, package declaration 7.1, package specification 7.1, parameter specification 6.1, record type 3.7, renaming declaration 8.5, subprogram body 6.3, subprogram declaration 6.1, type declaration 3.3.1, use clause 8.4, visibility 8.3, visible part 7.2 12

8.3 Visibility

The meaning of the occurrence of an identifier at a given place in the text is defined by the visibility rules and also, in the case of overloaded declarations, by the overloading rules. The identifiers considered in this chapter include any identifier other than a reserved word or an attribute designator. The places considered in this chapter are those where a lexical element (such as an identifier) occurs. The overloaded declarations considered in this chapter are those for subprograms and enumeration literals ♦. 1

For each identifier and at each place in the text (except for occurences as the entity indentifier in some form of declaration[52]), the visibility rules determine a set of declarations (with this identifier) that define possible meanings of an occurrence of the identifier. A declaration is said to be *visible* at a given place in the text when, according to the visibility rules, the declaration defines a possible meaning of this occurrence. Two cases arise. ♦ 2

- The visibility rules determine *at most one* possible meaning. In such a case the visibility rules are sufficient to determine the declaration defining the meaning of the occurrence of the identifier, or in the absence of such a declaration, to determine that the occurrence is not legal at the given point. ♦ 3
- The visibility rules determine *more than one* possible meaning. In such a case the occurrence of the identifier is legal at this point if and only if *exactly one* visible declaration is acceptable for the overloading rules in the given context (see section 6.6 for the rules of overloading and section 8.7 for the context used for overload resolution). 4

A declaration is only visible within a certain part of its scope; this part starts at the end of the declaration except in a package declaration [AI-00392], in which case it starts at the reserved word **is** given after the identifier of the package declaration. (This rule applies, in particular, for implicit declarations.) 5

Visibility is either by selection or direct. A declaration is visible *by selection* at places that are defined as follows. ♦ 6

(a) For a declaration given in the visible part of a package declaration: at the place of the selector after the dot of an expanded name whose prefix denotes the package. 7

(b) Omitted 8

(c) For a component ♦ declaration of a given record type declaration: at the place of the selector after the dot of a selected component whose prefix is appropriate for the type; also at the place of a component simple name (before the compound delimiter =>) in a named component association of 9

---

[52] This includes ending identifiers in subprogram bodies and packages.

an aggregate of the type.

(d) Omitted

(e) Omitted

(f) Omitted

♦

Where it is not visible by selection, a visible declaration is said to be *directly visible*. A declaration is directly visible within a certain part of its immediate scope; this part extends to the end of the immediate scope of the declaration, but excludes places where the declaration is hidden as explained below. In addition, a declaration occurring immediately within the visible part of a package can be made directly visible by means of a use clause according to the rules described in section 8.4. (See also section 8.6 for the visibility of library units.)

A declaration is said to be *hidden* within (part of) an inner declarative region if the inner region contains a homograph of this declaration; the outer declaration is then hidden within the immediate scope of the inner homograph. Each of two declarations is said to be a *homograph* of the other if both declarations have the same identifier and overloading is allowed for at most one of the two. If overloading is allowed for both declarations (i.e. they are both subprograms), then each of the two is a homograph of the other if they have the same identifier, operator symbol, or character literal, as well as the same parameter and result type profile (see 6.6).[53] Private type declarations and deferred constant declarations are hidden by their corresponding *full* declaration within the scope of the full declaration [AI-00385].

Within the specification of a subprogram, every declaration with the same designator as the subprogram is hidden; ♦ where hidden in this manner, a declaration is visible neither by selection nor directly.

Two declarations that occur immediately within the same declarative region must not be homographs except for private types and deferred constants and their respective full declarations as described above.

Whenever a declaration with a certain identifier is visible from a given point, the identifier and the declared entity (if any) are also said to be visible from that point. Direct visibility and visibility by selection are likewise defined for character literals♦. An operator is directly visible if and only if the corresponding operator declaration is directly visible. Finally, the notation associated with a basic operation is directly visible within the entire scope of this operation.

Example:

```
package P is
  package Q is
  end;
end;

package body P is
  A : BOOLEAN;
  B : BOOLEAN;
  package body Q is}
```

[53]Two enumeration literals cannot be homographs, because they are parameterless functions with different result types. [AI-00182]

```
      C : BOOLEAN;
      B : BOOLEAN;                    -- an inner homograph of B
    begin
      ...
      B := A;                         -- means Q.B := P.A;
      C := P.B;                       -- means Q.C := P.B;
    end;
  begin
    ...
    A := B;                           -- means P.A := P.B;
  end;
```

Note on the visibility of library units:

The visibility of library units is determined by with clauses (see 10.1.1) and by the fact that library units are implicitly declared in the package STANDARD (see 8.6). 20

Note on homographs:

The same identifier may occur in different declarations and may thus be associated with different entities, even if the scopes of these declarations overlap. Overlap of the scopes of declarations with the same identifier can result from overloading of subprograms and of enumeration literals. Such overlaps can also occur for entities declared in package visible parts and for ♦ record components, and parameters, where there is overlap of the scopes of the enclosing package declarations ♦, record type declarations, subprogram declarations, or renaming declarations. Finally overlapping scopes can result from nesting. 21

Note on immediate scope, hiding, and visibility:

The rules defining immediate scope, hiding, and visibility imply that a reference to an identifier within its own declaration is illegal (except for packages ♦). The identifier hides outer homographs within its immediate scope, that is, from the start of the declaration; on the other hand, the identifier is visible only after the end of the declaration. For this reason, all but the last of the following declarations are illegal: 22

```
K : INTEGER := K * K;              -- illegal
T : T;                             -- illegal
procedure P(X : P);                -- illegal
♦
procedure R(R : INTEGER♦);         -- an inner declaration is legal (although confusing)
```

References: aggregate 4.3, appropriate for a type 4.1, argument 2.8, basic operation 3.3.3, character literal 2.5, component association 4.3, component declaration 3.7, compound delimiter 2.2, declaration 3.1, declarative region 8.1, designate 3.8, enumeration literal specification 3.5.1, expanded name 4.1.3, extends 8.1, formal parameter 6.1, identifier 2.3, immediate scope 8.2, implicit declaration 3.1, lexical element 2.2, library unit 10.1, object 3.2, occur immediately within 8.1, operator 4.5, operator symbol 6.1, overloading 6.6 8.7, package 7, parameter 6.2, parameter association 6.4, parameter specification 6.1, program unit 6, record type 3.7, reserved word 2.9, scope 8.2, selected component 4.1.3, selector 4.1.3, simple name 4.1, subprogram 6, subprogram call 6.4, subprogram declaration 6.1, subprogram specification 6.1, type 3.3, type declaration 3.3.1, use clause 8.4, visible part 7.2 23

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
8.03	(01)	BI	CE	0010/06	88-12-30	The meaning of operations and identifiers in pragma arguments

8.03	(01)	ra	RE	0253/00	84-05-26	Are simple_names after "end" exempt from the visibility rules?
8.03	(02)	BI	RE	0541/00	87-08-05	What is the meaning of an identifier or name?
8.03	(03)	ra	RE	0134/00	83-11-07	use of undeclared identifiers not prohibited
8.03	(05)	BI	WA	0392/06	88-06-16	Visibility within a package declaration
8.03	(05)	ra	RE	0528/00	87-03-11	Does a self-referencing subprogram need a declaration?
8.03	(14)	ST	RE	0480/00	86-10-13	Visibility of predefined operators with derived types
8.03	(15)	BI	WA	0385/05	88-11-04	Hiding an incomplete, private, or deferred constant declaration
8.03	(15)	BI	WJ	0286/11	87-06-18	Declarations visible in a generic subprogram decl and body
8.03	(16)	na	CA	0512/02	87-03-17	Improve the example in 0370/05
8.03	(16)	ra	WJ	0370/06	87-01-13	Visibility of subprogam names within instantiations
8.03	(17)	BI	WA	0386/06	88-11-04	Incomplete, private, and deferred constant homographs
8.03	(17)	BI	WJ	0002/07	86-12-01	Deriving homographs for an enumeration literal and a function
8.03	(17)	BI	WJ	0012/06	88-05-23	Declaration of homographs by derivation and instantiation
8.03	(17)	BI	WJ	0330/12	86-07-23	Explicit declaration of enumeration literals
8.03	(18)	BI	WJ	0027/07	87-06-18	Visibility of type mark in explicit conversion or qualified expression

8.4 Use Clauses

A use clause achieves direct visibility of declarations that appear in the visible parts of named packages. 1

use_clause ::= **use** *package*_name {, *package*_name}; 2

For each use clause, there is a certain region of text called the *scope* of the use clause. This region starts immediately after the use clause. If a use clause is a declarative item of some declarative region, the scope of the clause extends to the end of the declarative region. If a use clause occurs within a context clause of a compilation unit, the scope of the use clause extends to the end of the declarative region associated with the compilation unit. 3

In order to define which declarations are made directly visible at a given place by use clauses, consider the set of packages named by all use clauses whose scopes enclose this place, omitting from this set any packages that enclose this place. A declaration that can be made directly visible by a use clause (a potentially visible declaration) is any declaration that occurs immediately within the visible part of a package of the set. A potentially visible declaration is actually made directly visible except in the following two cases: 4

- A potentially visible declaration is not made directly visible if the place considered is within the immediate scope of a homograph of the declaration. 5
- Potentially visible declarations that have the same identifier are not made directly visible unless each of them is either an enumeration literal specification or the declaration of a subprogram (by a subprogram declaration, a renaming declaration ♦, or an implicit declaration). 6

The elaboration of a use clause has no other effect. 7

Note:

The above rules guarantee that a declaration that is made directly visible by a use clause cannot hide an otherwise directly visible declaration. The above rules are formulated in terms of the set of packages named by use clauses. 8

Consequently, the following lines of text all have the same effect (assuming only one package P). 9

use P;
use P; **use** P, P;

Example of conflicting names in two packages: 10

```
package R is
  package TRAFFIC is
    type COLOR is (RED, AMBER, GREEN);
    ...
  end TRAFFIC;

  package WATER_COLORS is
    type COLOR is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
    ...
  end WATER_COLORS;

  use TRAFFIC;                  -- COLOR, RED, AMBER, and GREEN are directly visible
  use WATER_COLORS;             -- two homographs of GREEN are directly visible
                                -- but COLOR is no longer directly visible

  subtype LIGHT is TRAFFIC.COLOR;          -- Subtypes are used to resolve
  subtype SHADE is WATER_COLORS.COLOR;     -- the conflicting type name COLOR

  SIGNAL : LIGHT := GREEN;      -- that of TRAFFIC
  PAINT  : SHADE := GREEN;      -- that of WATER_COLORS

♦

end R;
```

Example of name identification with a use clause: 11

```
package D is
  T, U, V : BOOLEAN := FALSE;
end D;

with D;
package P is
  package E is
    B, W, V : INTEGER := 0;
  end E;

  procedure Q;
end P;

package body P is
  procedure Q is
    T, X : CHARACTER := 'a';
    use D, E;
  begin
    -- the name T means Q.T, not D.T
    -- the name U means D.U
    -- the name B means E.B
    -- the name W means E.W
    -- the name X means Q.X
    -- the name V is illegal : either D.V or E.V must be used
    ...
  end Q;
  ...
end P;
```

References: compilation unit 10.1, context clause 10.1, declaration 3.1, declarative item 3.9, declarative 12

region 8.1, direct visibility 8.3, elaboration 3.1 3.9, elaboration has no other effect 3.1, enumeration literal specification 3.5.1, extends 8.1, hiding 8.3, homograph 8.3, identifier 2.3, immediate scope 8.2, name 4.1, occur immediately within 8.1, package 7, scope 8.2, subprogram declaration 6.1, visible part 7.2

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
8.04	(01)	ST	RE	0274/00	84-08-27	Propposed extension of the USE clause - Record component visibility
8.04	(05)	BI	WJ	0286/11	87-06-18	Declarations visible in a generic subprogram decl and body
8.04	(05)	na	na	0156/01	86-01-24	Making some subprograms directly visible
8.04	(06)	na	na	0156/01	86-01-24	Making some subprograms directly visible

8.5 Renaming Declarations

A renaming declaration declares another name for an entity. 1

renaming_declaration ::= 2
identifier : type_mark **renames** *object*_name;
| ♦
| **package** identifier **renames** *package*_name;
| subprogram_specification **renames** *subprogram*_name;

The elaboration of a renaming declaration evaluates the name that follows the reserved word **renames** and thereby determines the entity denoted by this name (the renamed entity). At any point where a renaming declaration is visible, the identifier, or operator symbol} of this declaration denotes the renamed entity. The renamed item cannot be an attribute.[54] 3

The first form of renaming declaration is used for the renaming of objects. The renamed entity must be an object of the base type of the type mark. The properties of the renamed object are not affected by the renaming declaration. In particular, its value and whether or not it is a constant are unaffected; similarly, the constraints that apply to an object are not affected by renaming (any constraint implied by the type mark of the renaming declaration is ignored). The renaming declaration is legal only if exactly one object has this type and can be denoted by the object name. 4

♦ 5

The second form of renaming declaration is used ♦ for the renaming of ♦ packages. 6

The last form of renaming declaration is used for the renaming of subprograms ♦. The renamed subprogram ♦ and the subprogram specification given in the renaming declaration must have the same parameter and result type profile (see 6.6). The renaming declaration is legal only if exactly one visible subprogram ♦ satisfies the above requirements and can be denoted by the given subprogram ♦ name. In addition, parameter modes must be identical for formal parameters that are at the same parameter position. The subprogram name cannot be an attribute.[55] 7

The subtypes of the parameters and result (if any) of a renamed subprogram ♦ are not affected by renaming. These subtypes are those given in the original subprogram declaration ♦ (not those of the 8

---

[54]It is not obvious what form an attribute takes. Is it a constant or a function of no arguments? In addition, certain attributes cannot be renamed in Ada because there is no way to express their specifications (VAL and POS).

[55]T'VAL could not be renamed in Ada. In general it is undesirable to constrain an implementation to translate an attribute to a particular form, given that attributes are defined in such an *ad hoc* manner in Ada.

renaming declaration); even for calls that use the new name. ♦

A procedure can only be renamed as a procedure. A function can only be renamed as a function.[56] ♦ 9

Examples: 10

```
declare
  L : PERSON renames NEXT_PERSON;                    -- see 3.7
begin
  L.AGE := L.AGE + 1;
end;

♦
package TM renames TABLE_MANAGER;

function NO_FREE_SPACE (FOO : INTEGER) return BOOLEAN
    renames FREE_LIST_EMPTY;   --  see 8.5
♦
```

Example of a renaming declaration with new parameter names: 11

```
procedure ADD1 (N : in out INTEGER) renames INCREMENT; -- see 6.1
```

♦ 12

Notes:

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 points. ♦ 13

♦ 14

A subtype can be used to achieve the effect of renaming a type ♦ as in 16

```
♦ subtype MODE is TEXT_IO.FILE_MODE;
```

References: allow 1.6, attribute 4.1.4, base type 3.3, ♦ constant 3.2.1, constrained subtype 3.3, constraint 3.3, declaration 3.1, ♦ elaboration 3.1 3.9, enumeration literal 3.5.1, evaluation of a name 4.1, ♦ formal parameter 6.1, function 6.5, identifier 2.3, legal 1.6, mode 6.1, name 4.1, object 3.2, object declaration 3.2, operator 6.7, operator declaration 6.7, ♦ package 7, parameter 6.2, parameter specification 6.1, procedure 6.1, procedure call statement 6.4, reserved word 2.9, subcomponent 3.3, subprogram 6, subprogram call 6.4, subprogram declaration 6.1, subprogram specification 6.1, subtype 3.3.2, type 3.3, type mark 3.3.2, variable 3.2.1, visibility 8.3 17

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
8.05	(04)	BI	WJ	0001/10	86-07-23	Renaming and static expressions
8.05	(04)	ra	WI	0028/01	84-01-29	Can an object designated by an access value be renamed?
8.05	(05)	BI	RE	0738/00	88-12-21	At the end of the first sentence, insert "or a constant"
8.05	(05)	BI	WJ	0170/07	88-06-13	Renaming a slice

[56] Enumeration literals *cannot* be renamed as functions in AVA, since we require function specifications to have at least one parameter.

8.05	(05)	CR	WJ	0502/05	88-05-23	Error in 0170/06
8.05	(05)	ra	RE	0166/00	84-01-13	Rule is not clear
8.05	(07)	BI	WJ	0287/05	88-12-14	Resolving overloaded entry calls
8.05	(08)	ra	WJ	0245/08	88-05-23	Type conversion conformance for renamed subprogram/entry calls
8.05	(09)	co	RE	0221/00	84-03-13	Renaming of attributes defined as functions
8.05	(16)	ST	RE	0378/00	85-08-22	Subtype declarations as renamings

8.6 The Package Standard

The predefined types (for example the types BOOLEAN, CHARACTER and INTEGER) are the types that are declared in a predefined package called STANDARD; this package also includes the declarations of their predefined operations. The package STANDARD is described in Annex C. Apart from the predefined numeric types, the specification of the package STANDARD must be the same as or a superset of, the package described in Annex C, for all implementations of the language. 1

The package STANDARD forms a declarative region which encloses every library unit and consequently the main program; the declaration of every library unit is assumed to occur immediately within this package. The implicit declarations of library units are assumed to be ordered in such a way that the scope of a given library unit includes any compilation unit that mentions the given library unit in a with clause. However, the only library units that are visible within a given compilation unit are as follows: they include the library units named by all with clauses that apply to the given unit, and moreover, if the given unit is a secondary unit of some library unit, they include this library unit. 2

Notes:

♦ 3

The name of a library unit cannot be a homograph of a name that is already declared in package STANDARD [AI-00192]. 3a

If a type is declared in the visible part of a library package, then it is a consequence of the visibility rules that a basic operation (such as assignment) for this type is directly visible at places where the type itself is not visible (whether by selection or directly). However this operation can only be applied to operands that are visible and the declaration of these operands requires the visibility of either the type or one of its subtypes. 4

References: applicable with clause 10.1.1, ♦ block statement 5.6, declaration 3.1, declarative region 8.1, expanded name 4.1.3, hiding 8.3, identifier 2.3, implicit declaration 3.1, library unit 10.1, loop statement 5.5, main program 10.1, must 1.6, name 4.1, occur immediately within 8.1, operator 6.7, package 7, program unit 6, secondary unit 10.1, subtype 3.3, type 3.3, visibility 8.3, with clause 10.1.1 5

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
8.06	(02)	ra	WJ	0192/05	88-05-23	Allowed names of library units

8.7 The Context of Overload Resolution

Overloading is defined for subprograms, enumeration literals, operators, ♦ and also for the operations that are inherent in several basic operations such as assignment, membership tests, ♦ aggregates, and string literals. 1

For overloaded entities, overload resolution determines the actual meaning that an occurrence of an identifier or character literal [AI-00352] has, whenever the visibility rules have determined that more than one meaning is acceptable at the place of this occurrence; overload resolution likewise determines the actual meaning of an occurrence of an operator or some basic operation. 2

At such a place all visible declarations are considered. The occurrence is only legal if there is exactly one interpretation of each constituent of the innermost *enclosing* complete context; a *complete context* is one of the following: 3

- A declaration. 4
- A statement. 5
- ♦ 6

When considering possible interpretations of a complete context, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form described below. 7

(a) Any rule that requires a name or expression to have a certain type, or to have the same type as another name or expression. 8

(b) Any rule that requires the type of a name or expression to be a type of a certain class; similarly, any rule that requires a certain type to be a discrete, integer, ♦, universal, character, or boolean type. 9

(c) Any rule that requires a prefix to be appropriate for a certain type. 10

(d) Any rule that specifies a certain type as the result type of a basic operation, and any rule that specifies that this type is of a certain class. 11

(e) The rules that require the type of an aggregate or string literal to be determinable solely from the enclosing complete context (see 4.3 and 4.2). Similarly, the rules that require the type (e.g. meaning [AI-00462]) of the prefix of an attribute, the type of the expression of a case statement, or the type of the operand of a type conversion, to be determinable independently of the context (see 4.1.4, 5.4, 4.6, and 6.4.1). Any overloaded identifiers occurring in the argument for 'FIRST(N), 'LAST(N), and 'RANGE(N) must be resolved independently of the context in which these attributes are used [AI-00193]. 12

(f) The rules given in section 6.6, for the resolution of overloaded subprogram calls; in section 4.6, for the implicit conversions of universal expressions; in section 3.6.1, for the interpretation of discrete ranges with bounds having a universal type; and in section 4.1.3, for the interpretation of an expanded name whose prefix denotes a subprogram. 13

♦ 14

Similarly, the simple names given in context clauses (see 10.1.1) ♦ follow different rules.[57] 15

Notes:

[57]By [AI-00297] parent unit names in subunits (see ARM 10.2) also follow different rules, but subunits are excluded from the AVA subset.

If there is only one possible interpretation, the identifier denotes the corresponding entity. However, this does not mean that the occurrence is necessarily legal since other requirements exist which are not considered for overload resolution; for example, the fact that an expression is static, the parameter modes, whether an object is constant, conformance rules, ♦ order of elaboration, and so on. 16

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). 17

A loop parameter specification is a declaration, and hence a complete context. 18

Rules that require certain constructs to have the same parameter and result type profile fall under the category (a) ♦[58]. 19

References: aggregate 4.3, assignment 5.2, basic operation 3.3.3, case statement 5.4, class of type 3.3, declaration 3.1, enumeration literal 3.5.1, exception 11, expression 4.4, formal part 6.1, identifier 2.3, legal 1.6, literal 4.2, loop parameter specification 5.5, membership test 4.5.2, name 4.1, operation 3.3.3, operator 4.5, overloading 6.6, ♦ statement 5, static expression 4.9, static subtype 4.9, subprogram 6, subtype 3.3, type conversion 4.6, visibility 8.3 20

Rules of the form (*a*): assignment 5.2, choice 3.7.3 4.3.2 5.4, component association 4.3.1 4.3.2, discrete range 3.6.1 5.5 9.5, index constraint 3.6.1, index expression 4.1.1 4.1.2 9.5, initial value 3.2.1, membership test 4.5.2, parameter association 6.4.1, parameter and result type profile 8.5 12.3.6, qualified expression 4.7, range constraint 3.5, renaming of an object 8.5, result expression 5.8 21

Rules of the form (*b*): assignment 5.2, case expression 5.4, condition 5.3 5.5 5.7 9.7.1, discrete range 3.6.1 5.5 9.5, integer type declaration 3.5.4, membership test 4.4, number declaration 3.2.2, selected component 4.1.3, short-circuit control form 4.4, val attribute 3.5.5 22

Rules of the form (*c*): indexed component 4.1.1, selected component 4.1.3 23

Rules of the form (*d*): aggregate 4.3, membership test 4.4, numeric literal 2.4, short-circuit control form 4.4, string literal 4.2 24

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
8.07	(02)	BI	WA	0352/04	88-06-16	Character literals are subject to overload resolution
8.07	(03)	BI	CE	0297/04	88-11-20	Resolving overloaded parent unit names
8.07	(03)	BI	WI	0117/01	87-04-16	"Context" means "innermost enclosing complete context"
8.07	(03)	co	WJ	0120/05	86-12-01	Overload resolution for assignment statements
8.07	(07)	BI	CE	0297/04	88-11-20	Resolving overloaded parent unit names
8.07	(07)	BI	WJ	0157/05	86-07-23	Overloading resolution and parenthesized expressions
8.07	(08)	ra	WJ	0193/05	86-07-23	The value of ’FIRST’s argument in overloading resolution
8.07	(12)	BI	CA	0462/03	88-10-03	The "meaning" vs. the "type" of attribute prefixes
8.07	(12)	ra	CA	0481/04	88-10-03	Overload resolution for the operand of a type conversion
8.07	(13)	BI	WI	0119/02	88-03-28	The prefix of an expanded name
8.07	(13)	BI	WJ	0287/05	88-12-14	Resolving overloaded entry calls
8.07	(19)	ra	CA	0183/05	88-10-03	Conformance rules are not used for overload resolution
8.07	(19)	ra	CE	0463/03	88-12-08	The number of parameters is used for overload resolution
8.07	(21)	ra	CA	0183/05	88-10-03	Conformance rules are not used for overload resolution

[58]The text "the same holds for rules that require conformance of two constructs since conformance requires that corresponding names be given the same meaning by the visibility and overloading rules" is deleted as per [AI-00183].

Chapter 9

TASKS: REMOVED

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
9.01	(02)	BI	WA	0359/04	88-11-04	Equivalence of single task declarations
9.03	(03)	BI	RE	0283/01	86-07-10	When is the activation of a task concluded if it fails its activation?
9.03	(03)	ra	WJ	0268/06	86-07-23	Activation of already abnormal tasks
9.03	(04)	BI	WJ	0198/09	88-05-23	Termination of unactivated tasks
9.03	(04)	ra	RE	0440/00	86-07-10	"direct" or "indirect" task creation
9.03	(05)	BI	WJ	0237/06	86-07-23	Instances having implicit package bodies
9.03	(08)	BI	WJ	0198/09	88-05-23	Termination of unactivated tasks
9.04	(00)	co	WJ	0167/04	86-07-23	It is possible to access a task from outside its master
9.04	(05)	BI	WJ	0173/05	86-12-01	Completion of execution by exception propagation
9.04	(06)	ra	WJ	0441/06	88-05-23	A task without dependents can be completed but not terminated
9.04	(13)	BI	CA	0399/13	88-10-03	Status of library tasks when the main program terminates
9.05	(00)	co	RE	0373/00	85-08-04	Must there be a corresponding accept for each entry?
9.05	(03)	ra	CE	0369/06	88-11-20	Representing values of discrete base type
9.05	(05)	BI	WJ	0287/05	88-12-14	Resolving overloaded entry calls
9.05	(08)	ST	RE	0214/00	84-03-13	Allow accept statements in program units nested in tasks
9.06	(01)	BI	WJ	0464/05	88-05-23	Delay statements executed by the environment task
9.06	(01)	ra	WJ	0201/07	88-05-23	The relation between TICK, CLOCK, and the delay statement
9.06	(04)	ra	WJ	0201/07	88-05-23	The relation between TICK, CLOCK, and the delay statement
9.06	(04)	ST	RE	0223/00	84-03-13	Resolution for the function CLOCK
9.06	(05)	BI	RE	0754/00	88-12-21	TIME is a type, and has no value
9.06	(05)	ra	RE	0194/00	84-03-13	Function CALENDAR.CLOCK
9.06	(05)	ra	WJ	0195/09	88-05-23	The intended use of CLOCK
9.06	(05)	ra	WJ	0201/07	88-05-23	The relation between TICK, CLOCK, and the delay statement
9.06	(06)	ra	WJ	0196/05	86-07-23	Use of 86_400.0 in TIME_OF
9.06	(07)	ra	WJ	0355/06	86-12-01	Pragma ELABORATE for predefined library packages
9.06	(07)	ST	RE	0442/00	86-07-10	Time zone information in package CALENDAR
9.07.01	(04)	ra	RE	0029/01	84-03-13	How often are conditions evaluated?
9.07.01	(05)	ra	WJ	0030/07	86-07-23	All guards need not be evaluated first
9.07.01	(06)	ra	RE	0443/02	86-10-13	Is a rendezvous "on the fly" possible?
9.07.01	(06)	ra	WP	0233/11	88-12-07	Effect of priorities in selective waits
9.07.02	(01)	ra	WJ	0276/07	87-02-23	Rendezvous that are "immediately possible" vs. timed entry calls
9.07.02	(01)	ra	WJ	0444/05	87-08-06	Conditional entry calls can be queued momentarily
9.07.03	(04)	ra	RE	0445/00	86-07-10	Cancelling a timed entry call
9.07.03	(04)	ra	WJ	0276/07	87-02-23	Rendezvous that are "immediately possible" vs. timed entry calls
9.08	(00)	ra	WJ	0197/07	86-12-01	With SYSTEM clause not needed for pragma PRIORITY
9.08	(01)	co	WJ	0031/06	86-07-23	Out-of-range argument to pragma PRIORITY
9.08	(01)	ra	WI	0042/02	87-04-20	Effect of recompiling SYSTEM on pragma PRIORITY
9.08	(02)	BI	CA	0548/01	88-10-03	Effect of pragma PRIORITY on library unit elaboration
9.08	(04)	BI	RE	0594/00	88-10-05	Problems with Preemptive Scheduling
9.08	(04)	co	WI	0033/02	84-03-26	Effect of priorities on calls queued for an entry
9.08	(04)	NB	WJ	0288/06	86-07-23	Effect of priorities during activation
9.08	(04)	ra	WJ	0032/09	87-03-16	Preemptive scheduling is required
9.08	(04)	ra	WP	0233/11	88-12-07	Effect of priorities in selective waits

9.09	(05)	ST	RE	0529/00	87-03-11	Resolving the meaning of an attribute name
9.09	(06)	ra	WJ	0034/06	86-07-23	Value of COUNT in an accept statement
9.10	(04)	ra	WA	0360/05	88-04-12	Abort of several tasks
9.10	(05)	BI	WJ	0198/09	88-05-23	Termination of unactivated tasks
9.10	(06)	BI	RE	0224/02	88-09-02	Synchronization point definition.
9.10	(06)	BI	WJ	0446/05	87-09-25	Raising an exception in an abnormally completed task
9.10	(08)	BI	RE	0581/00	88-08-31	Abort and undefined variables
9.11	(04)	NB	WI	0004/01	83-10-14	Packed composite objects and shared variables
9.11	(04)	ST	WI	0142/02	84-08-27	Proposed solution to packed composite object/shared variable problem
9.11	(04)	BI	RE	0106/00	83-11-07	"two synchronization" to "two successive synchronization"
9.11	(05)	BI	RE	0106/00	83-11-07	"two synchronization" to "two successive synchronization"
9.11	(05)	NB	WI	0004/01	83-10-14	Packed composite objects and shared variables
9.11	(05)	ST	WI	0142/02	84-08-27	Proposed solution to packed composite object/shared variable problem
9.11	(07)	ra	RE	0353/01	88-09-02	Shared variables.
9.11	(09)	BI	WI	0141/03	84-12-10	Meaning of "update" for shared actual parameters
9.11	(10)	ST	WI	0142/02	84-08-27	Proposed solution to packed composite object/shared variable problem
9.11	(11)	BI	WI	0447/02	86-12-15	Indivisible semantic effects

Chapter 10

PROGRAM STRUCTURE AND COMPILATION ISSUES

The overall structure of programs and the facilities for separate compilation are described in this chapter. A program is a collection of one or more compilation units submitted to a compiler in one or more compilations. Each compilation unit specifies the separate compilation of a construct which can be a subprogram declaration or body or a package declaration or body[59] ♦. 1

References: compilation 10.1, compilation unit 10.1, package body 7.1, package declaration 7.1, subprogram body 6.3, subprogram declaration 6.1 2

10.1 Compilation Units - Library Units

The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation units. 1

```
compilation ::= {compilation_unit}                                    2

compilation_unit ::=
    context_clause library_unit | context_clause secondary_unit

library_unit ::=
    subprogram_declaration | package_declaration | ♦
  | ♦ subprogram_body

secondary_unit ::= library_unit_body | ♦

library_unit_body ::= subprogram_body | package_body
```

The compilation units of a program are said to belong to a *program library*. A compilation unit defines either a library unit or a secondary unit. A secondary unit is ♦ the ♦ separately compiled proper body of a library unit ♦. The ♦ designator of a separately compiled subprogram ♦ must be an identifier. Within a program library the simple names of all library units must be distinct identifiers. 3

The effect of compiling a library unit is to define ♦ this unit as one that belongs to the program library. For the visibility rules, each library unit acts as a declaration that occurs immediately within the package STANDARD. 4

---

[59] As a logical consequence of this, we require, contrary to [AI-507], that in the absence of compiler detected errors, there should be no difference between compiling $file_1$ = ''$Unit_1$'' then $file_2$ = ''$Unit_2$'' vs. compiling $file_3$ = ''$Unit_1$ $Unit_2$''. See 10.3.

The effect of compiling a secondary unit is to define the body of a library unit ♦. 5

A subprogram body given in a compilation unit is interpreted as a secondary unit if the program library already contains a library unit that is a subprogram declaration [AI-00199] with the same name; it is otherwise interpreted both as a library unit and as the corresponding library unit body (that is, as a secondary unit).[60] 6

The compilation units of a compilation are compiled in the given order. ♦ 7

A subprogram that is a library unit can be used as a *main program* in the usual sense. Each main program acts as if called by some environment task; the means by which this execution is initiated are not prescribed by the language definition. An implementation may impose certain requirements on the parameters and on the result, if any, of a main program (these requirements must be stated in Appendix F). In any case, every implementation is required to allow, at least, main programs that are parameterless procedures, and every main program must be a subprogram that is a library unit. 8

Notes:

A simple program may consist of a single compilation unit. ♦ 9

♦ Two library subprograms must have distinct simple 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.[61] ♦ 10

References: allow 1.6, context clause 10.1.1, declaration 3.1, designator 6.1, environment 10.4, hiding 8.3, identifier 2.3, library unit 10.5, local declaration 8.1, must 1.6, name 4.1, occur immediately within 8.1, ♦ overloading 6.6 8.7, package body 7.1, package declaration 7.1, parameter of a subprogram 6.2, ♦ procedure 6.1, program unit 6, proper body 3.9, renaming declaration 8.5, simple name 4.1, standard package 8.6, subprogram 6, subprogram body 6.3, subprogram declaration 6.1, visibility 8.3 11

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
10.01	(03)	ra	WJ	0418/06	87-08-06	Self-referencing with clauses
10.01	(06)	BI	WJ	0199/08	86-07-23	Implicit declaration of library subprograms
10.01	(06)	BI	WJ	0225/09	86-07-23	Secondary units for generic subprograms
10.01	(06)	na	na	0254/01	85-02-04	[combined with 0225]
10.01	(06)	ra	WJ	0266/09	86-07-23	A body cannot be compiled for a library unit instantiation
10.01	(08)	BI	WA	0513/08	88-11-04	A main program can be given by a generic instantiation
10.01	(08)	BI	WI	0222/08	87-10-29	Executing a program
10.01	(08)	co	RE	0435/00	86-06-19	A subprogram cannot be a library unit.

[60] Its interpretation as a library unit does not result in the creation of a library unit *that is a subprogram declaration*.

[61] The expanded name STANDARD.L cannot be used for a library unit L unless the name STANDARD is visible and we are in compilation unit L or the current compilation unit contained L in a with clause. See 8.6.

10.1.1 Context Clauses - With Clauses

A context clause is used to specify the library units whose names are needed within a compilation unit. 1

context_clause ::= {with_clause {use_clause}} 2

with_clause ::= **with** *unit*_simple_name {, *unit*_simple_name};

The names that appear in a context clause must be the simple names of library units other than the one to which this context clause is attached.[62] The simple name of any library unit is allowed within a with clause. The only names allowed in a use clause of a context clause are the simple names of library packages mentioned by previous with clauses of the context clause. A simple name declared by a renaming declaration is not allowed in a context clause. 3

The with clauses and use clauses of the context clause of a library unit *apply* to this library unit and also to the secondary unit that defines the corresponding body (whether such a clause is repeated or not for this unit). Similarly, the with clauses and use clauses of the context clause of a compilation unit *apply* to this unit ♦. 4

If a ♦ library unit is named by a with clause that applies to a compilation unit, then this library unit is directly visible within the compilation unit, except where hidden; the library unit is visible as if declared immediately within the package STANDARD (see 8.6). 5

♦ Dependences among compilation units are defined by with clauses; that is, a compilation unit *depends* on those library units that are mentioned in with clauses that apply to it. These dependences between units are taken into account for the determination of the allowed order of compilation (and recompilation) of compilation units, as explained in section 10.3, and for the determination of the allowed order of elaboration of compilation units, as explained in section 10.5. Circular dependencies between library *units* are not allowed. A legal compilation must not permit ambiguity in the order of elaboration of compilation units **where this ambiguity could lead to an ambiguity in the value of a variable when the main program is elaborated**. 6

Notes:

A library unit named by a with clause of a compilation unit is visible (except where hidden) within the compilation unit and hence can be used as a corresponding program unit. Thus within the compilation unit, 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. 7

The ♦ rules given for with clauses are such that the same effect is obtained whether the name of a library unit is mentioned once or more than once by the applicable with clauses, or even within a given with clause. 8

Example 1 : A main program:

---

[62] By [AI-00418] the only legitimate self-reference in a with clause is

```
with P2;
procedure P2 is ... end;
```

where a subprogram declaration of P2 exists as a library unit. AVA disallows even this.

The following is an example of a main program consisting of a single compilation unit: a procedure for printing the integer approximation of the ♦ roots of a quadratic equation. The predefined package AVA_IO and a user-defined package INTEGER_OPERATIONS (containing the definition of the type INT and of the packages INTEGER_IO and INTEGER_FUNCTIONS) are assumed to be already present in the program library. Such packages may be used by other main programs. 9

10

```
with AVA_IO, INTEGER_OPERATIONS; use INTEGER_OPERATIONS;
procedure QUADRATIC_EQUATION is
   IA, IB, IC : string (1..6) := "123456";
   A, B, C, D : INTEGER := 0;
   use AVA_IO,                      -- achieves direct visibility of PUT and EOL
       INTEGER_FUNCTIONS;           -- achieves direct visibility of ISQRT
begin
   GET(standard_input,IA); GET(standard_input,IB); GET(standard_input,IC);
   A := integer'value(IA);
   B := integer'value(IB);
   C := integer'value(IB);
   D := B**2 - 4*A*C;
   if D < 0 then
      PUT(standard_output,"Imaginary Roots.");
   else
      PUT(standard_output,"Integer Roots : X1 = ");
      PUT(standard_output,integer'image((-B - ISQRT(D))/(2*A)));
      PUT(standard_output," X2 = ");
      PUT(standard_output,integer'image((-B + ISQRT(D))/(2*A)));
   end if;
   PUT(standard_output,EOL);
end QUADRATIC_EQUATION;
```

Notes on the example:

The with clauses of a compilation unit need only mention the names of those library subprograms and packages whose visibility is actually necessary within the unit. They need not (and should not) mention other library units that are used in turn by some of the units named in the with clauses, unless these other library units are also used directly by the current compilation unit. For example, the body of the package INTEGER_OPERATIONS may need elementary operations provided by other packages. The latter packages should not be named by the with clause of QUADRATIC_EQUATION since these elementary operations are not directly called within its body. 11

References: allow 1.6, compilation unit 10.1, direct visibility 8.3, elaboration 3.9, hiding 8.3, instance 12.3, library unit 10.1, main program 10.1, must 1.6, name 4.1, package 7, package body 7.1, package declaration 7.1, procedure 6.1, program unit 6, secondary unit 10.1, simple name 4.1, standard predefined package 8.6, subprogram body 6.3, subprogram declaration 6.1, type 3.3, use clause 8.4, visibility 8.3 12

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
10.01.01	(04)	BI	WJ	0226/06	86-07-23	Applicability of context clauses to subunits

10.1.2 Examples of Compilation Units

A compilation unit can be split into a number of compilation units. For example, consider the following program.

```
package PROCESSOR is

   SMALL : constant := 20;
   TOTAL : INTEGER  := 0;

   package STOCK is
      LIMIT : constant := 1000;
      type tarray is array (1 .. LIMIT) of INTEGER;
      TABLE :  TARRAY := (others => 0);
      procedure RESTART;
   end STOCK;

   procedure UPDATE(X : INTEGER);

end PROCESSOR;

package body PROCESSOR is

   package body STOCK is
      procedure RESTART is
      begin
         for N in 1 .. LIMIT loop
            TABLE(N) := N;
         end loop;
      end;
   begin
      RESTART;
   end STOCK;

   procedure UPDATE(X : INTEGER) is
      use STOCK;
   begin
      ...
      TABLE(X) := TABLE(X) + SMALL;
      ...
   end UPDATE;

begin
   ...
   STOCK.RESTART;  -- reinitializes TABLE
   ...
end PROCESSOR;
```

The following four compilation units define a program with an effect equivalent to the above example (the broken lines between compilation units serve to remind the reader that these units need not be contiguous texts).

Example 2 : Several compilation units:

```
package STOCK is
   LIMIT : constant := 1000;
   type tarray is array (1 .. LIMIT) of INTEGER;
```

```
    TABLE : TARRAY := (others => 0);
  procedure RESTART;
end STOCK;

-----------------------------------------------

package body STOCK is                                          6
  procedure RESTART is
  begin
    for N in 1 .. LIMIT loop
      TABLE(N) := N;
    end loop;
  end;
begin
  RESTART;
end STOCK;

-----------------------------------------------

with STOCK;                                                    7
package PROCESSOR is

   SMALL : constant := 20;
   TOTAL : INTEGER  := 0;

   procedure UPDATE(X : INTEGER);

end PROCESSOR;

package body PROCESSOR is

   procedure UPDATE(X : INTEGER) is
     use STOCK;
   begin
     ...
     TABLE(X) := TABLE(X) + SMALL;
     ...
   end UPDATE;

begin
   ...
   STOCK.RESTART;  --  reinitializes TABLE
   ...
end PROCESSOR;
```

Note that in the latter version, the package STOCK has no visibility of outer identifiers other than the predefined identifiers (of the package STANDARD). In particular, STOCK does not use any identifier declared in PROCESSOR such as SMALL or TOTAL; otherwise STOCK could not have been extracted from PROCESSOR in the above manner. The package PROCESSOR, on the other hand, depends on STOCK and mentions this package in a with clause. This permits the inner occurrences of STOCK in the expanded name STOCK.RESTART and in the use clause. 8

These three compilation units can be submitted in one or more compilations. For example, it is possible to submit the package specification and the package body together and in this order in a single compilation. 9

References: compilation unit 10.1, declaration 3.1, identifier 2.3, package 7, package body 7.1, package specification 7.1, program 10, standard package 8.6, use clause 8.4, visibility 8.3, with clause 10.1.1

10.2 Subunits of Compilation Units : Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
10.02	(03)	ra	WJ	0035/06	86-12-04	Body stubs are not allowed in package specifications
10.02	(05)	BI	CE	0297/04	88-11-20	Resolving overloaded parent unit names
10.02	(05)	BI	WA	0482/04	88-11-04	Ancestor library unit for secondary units and subunits
10.02	(05)	BI	WJ	0289/05	86-07-23	Ancestor unit names in separate clauses must be simple names
10.02	(05)	ST	RE	0458/00	86-09-05	Problem with naming of subunits
10.02	(05)	ST	RE	0572/00	88-07-06	Unique path name for subunits

10.3 Order of Compilation

The rules defining the order in which units can be compiled are direct consequences of the visibility rules and, in particular, of the fact that any library unit that is mentioned by the context clause of a compilation unit is visible in the compilation unit.[63]

A compilation unit must be compiled after all library units named by its context clause. A secondary unit that is a subprogram or package body must be compiled after the corresponding library unit. ♦

If any illegal construct [AI-00261] is detected while attempting to compile a compilation unit, then the attempted compilation of that compilation unit is rejected and it has no effect whatsoever on the program library; the same holds for recompilations (no compilation unit can become obsolete because of such a recompilation).

The order in which the compilation units of a program are compiled must be consistent with the partial ordering defined by the above rules.

Similar rules apply for recompilations. A compilation unit is potentially affected[64] by a change in any library unit named by its context clause. A secondary unit is potentially affected by a change in the corresponding library unit or in the context clause of the corresponding library unit. If a compilation unit is successfully recompiled, the compilation units potentially affected by this change are obsolete and must be recompiled unless they are no longer needed. An implementation may be able to reduce the compilation costs if it can deduce that some of the potentially affected units are not actually affected by the change[65].

♦ Changes in a subprogram or package body do not affect other compilation units (in the sense of not requiring recompilation) since these compilation units only have access to the subprogram or package

---

[63] See also 10.1.1 and 10.5.

[64] By *potentially effected* we mean that it may require recompilation.

[65] **IMPLEMENTATION REQUIREMENT**: [AI-507] states that if one compilation unit is a replacement for another in the same compilation, an implementation can reject the compilation in whole or in part. AVA does not allow this, but rather requires replacement of the first compilation unit by the second, with obsolescence of dependents of the first compilation unit. Worries about this compiler dependency can be completely avoided if all compilations are single compilation units.

specification. An implementation is only allowed to deviate from this rule ♦ for certain compiler optimizations ♦ as described below.

- ♦ 7
- For optimization purposes, an implementation may compile several units of a given compilation in a way that creates further dependences among these compilation units. The compiler must then take these dependences into account when deciding on the need for recompilations. It is *not* implementation dependent whether a compilation unit is obsolete. The additional information computed by a compiler may make the recompilation of the obsolescent unit much faster. For instance, it may detect that the the change to the governing unit really has no effect on the dependent unit. In which case, recompilation of the obsolescent dependent merely consists of marking it ‘‘not obsolescent’’. See [AI-530]. 8
- ♦ 9

Examples of Compilation Order: 10

(a) In example 1 (see 10.1.1): The procedure QUADRATIC_EQUATION must be compiled after the library packages AVA_IO and INTEGER_OPERATIONS since they appear in its with clause. 11

(b) In example 2 (see 10.1.2): The package body STOCK must be compiled after the corresponding package specification. 12

(c) In example 2 (see 10.1.2): The specification of the package STOCK must be compiled before the package PROCESSOR. On the other hand, the package PROCESSOR can be compiled either before or after the package body STOCK. 13

(d) Omitted 14

(e) Omitted 15

Notes:

For library packages, it follows from the recompilation rules that a package body is made obsolete by the recompilation of the corresponding specification. If the new package specification is such that a package body is not required (that is, if the package specification does not contain the declaration of a program unit), then the recompilation of a body for this package is not required. In any case, the obsolete package body must not be used and can therefore be deleted from the program library.[66] 16

References: compilation 10.1, compilation unit 10.1, context clause 10.1.1, elaboration 3.9, library unit 10.1, local declaration 8.1, name 4.1, package 7, package body 7.1, package specification 7.1, procedure 6.1, procedure body 6.3, proper body 3.9, subprogram body 6.3, subprogram declaration 6.1, subprogram specification 6.1, type 3.3, variable 3.2.1, visibility 8.3, with clause 10.1.1 17

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
10.03	(03)	BI	WA	0256/23	88-06-16	"Successful" compilation
10.03	(03)	BI	WJ	0261/03	86-07-23	Change "any error" to "any illegal construct"
10.03	(03)	na	na	0118/03	84-09-03	[combined with 0255]
10.03	(03)	ra	WB	0255/07	88-06-15	Partial processing of compilation files
10.03	(05)	BI	CB	0506/04	88-11-08	Instantiation before occurrence of a body
10.03	(05)	BI	RE	0602/00	88-11-08	Instantiation when no later body is compiled
10.03	(05)	ra	WA	0530/04	88-06-16	Recompilation of unused with’d units
10.03	(05)	ra	WI	0400/07	87-03-11	Obsolete package bodies
10.03	(06)	BI	WJ	0408/11	87-08-20	Effect of compiling generic unit bodies separately

[66] See [AI-400] and 10.5.

10.03	(06)	NB	WI	0465/03	87-03-18	Pragma INLINE_GENERIC
10.03	(07)	BI	WJ	0200/08	86-12-01	Dependences created by inline of generic instantiations
10.03	(08)	BI	WA	0507/03	88-11-04	Duplicate units in a compilation
10.03	(08)	na	na	0323/11	87-02-04	[replaced by 0506 and 0507]
10.03	(09)	ra	WJ	0257/04	86-07-23	Restricting generic unit bodies to compilations

10.4 The Program Library

Compilers are required to enforce the language rules in the same manner for a program consisting of several compilation units ♦ as for a program submitted as a single compilation. Consequently, a library file containing information on the compilation units of the program library must be maintained by the compiler or compiling environment. This information may include symbol tables and other information pertaining to the order of previous compilations. 1

A normal submission to the compiler consists of the compilation unit(s) and the library file. The latter is used for checks and is updated for each compilation unit successfully compiled.[67] 2

Notes:

A single program library is implied for the compilation units of a compilation. The possible existence of different program libraries and the means by which they are named are not concerns of the language definition; they are concerns of the programming environment. 3

There should be commands for creating the program library of a given program or of a given family of programs. These commands may permit the reuse of units of other program libraries. Finally, there should be commands for interrogating the status of the units of a program library. The form of these commands is not specified by the language definition. 4

References: compilation unit 10.1, context clause 10.1.1, order of compilation 10.3, program 10.1, program library 10.1 ♦, use clause 8.4, with clause 10.1.1 5

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
10.04	(02)	BI	WA	0256/23	88-06-16	"Successful" compilation

10.5 Elaboration of Library Units

Before the execution of a main program, all library units needed by the main program(including the subprogram used as the main program [AI-00158]) are elaborated, as well as the corresponding library unit bodies, if any. The library units needed by the main program are: those named by with clauses applicable to the main program ♦ and to its body; ♦ those named by with clauses applicable to these library units themselves, to the corresponding library unit bodies ♦; and so on, in a transitive manner. If a package specification needed by a main program does not require a package body, then a body for the package is elaborated only if a (non-obsolete) package body exists in the program library when an attempt is made to execute the main program; if such a package body has been 1

[67] In the case of a file consisting of multiple compilation units it is permitted to reject the entire file if one of the compilation units is illegal. This is in line with the permissive view taken in [AI-00255]. It is a necessary side effect of different possibile compilation strategies. If the entire file is checked syntactically before any static semantic checks are done, then no program unit has been successfully analyzed when a syntactic error is encountered.

compiled previously but is obsolete when the attempt is made to execute the main program, it is not elaborated. It is highly recommended that an implementation warn programmers that an unneeded package body has been made obsolete so such bodies are not unintentionally omitted when the main program is executed [AI-00400].

The elaboration of these library units and of the corresponding library unit bodies is performed in an order consistent with the partial ordering defined by the with clauses (see 10.3). ♦ 2

It is required that when a library unit has a corresponding secondary unit, the secondary unit is elaborated *immediately* after the library unit.[68] ♦ 3

♦ 4

The program is illegal if no consistent order can be found (that is, if a circularity exists). The elaboration of the compilation units of the program is performed in some order that is otherwise not defined by the language. 5

References: allow 1.6, compilation unit 10.1, context clause 10.1.1, dependence between compilation units 10.3, elaboration 3.9, illegal 1.6, in some order 1.6, library unit 10.1, name 4.1, main program 10.1, secondary unit 10.1, separate compilation 10.1, simple name 4.1, with clause 10.1.1 6

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
10.05	(01)	BI	WJ	0158/05	88-05-23	The main program is elaborated before it is called
10.05	(02)	BI	RE	0772/00	88-12-21	Incomplete
10.05	(02)	BI	WJ	0113/12	88-05-23	A subunit's with clause can name its ancestor library unit
10.05	(02)	ra	WJ	0354/03	86-12-01	On the elaboration of library units
10.05	(04)	BI	WJ	0236/12	88-12-14	Pragma ELABORATE for bodiless packages with tasks
10.05	(04)	ra	WJ	0298/05	86-12-01	Interaction between pragmas ELABORATE and INTERFACE
10.05	(04)	ra	WJ	0355/06	86-12-01	Pragma ELABORATE for predefined library packages
10.05	(04)	ST	RE	0421/00	86-05-05	Eliminate pragma ELABORATE

10.6 Program Optimization

Optimization of the elaboration of declarations and the execution of statements may be performed by compilers. In particular, a compiler may be able to optimize a program by evaluating certain expressions, in addition to those that are static expressions. Should one of these expressions, whether static or not, be such that an exception would be raised by its evaluation, then the code in that path of the program can be replaced by code to raise the exception; the same holds for exceptions raised by the evaluation of names and simple expressions ♦. 1

A compiler may find that some statements or subprograms will never be executed, for example, if their execution depends on a condition known to be FALSE. The corresponding object machine code can then 2

---

[68] **IMPLEMENTATION REQUIREMENT**: This avoids certain erroneous programs based on order of elaboration. See [AI-00354]. It has some other effects.

(a) It rules out mutual recursion between routines defined in different packages.

(b) If package specification B depends on package A, then the package body of A (if there is one) is elaborated before package specification B.

(c) Obviously, a package specification is always elaborated before its body.

be omitted. This rule permits the effect of *conditional compilation* within the language.

Note:

An expression whose evaluation is known to raise an exception need not represent an error if it occurs in a statement or subprogram that is never executed. The compiler may warn the programmer of a potential error. 3

References: condition 5.3, declaration 3.1, elaboration 3.9, evaluation 4.5 , exception 11 , expression 4.4, false boolean value 3.5.3, program 10, raising of exceptions 11.3, statement 5, static expression 4.9, subprogram 6 4

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
10.06	(01)	ra	RE	0531/00	87-03-11	exceptions and static expressions

Chapter 11
EXCEPTIONS

This chapter defines the facilities for dealing with errors or other exceptional situations that arise during program execution. Such a situation is called an *exception*. To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Executing some actions, in response to the arising of an exception, is called *handling* the exception. 1

An exception declaration declares a name for an exception. There is no facility in AVA for *programmer defined* exceptions. An exception can be raised by a raise statement, or it can be raised by another statement or operation that *propagates* the exception. When an exception arises, control is transferred to a user-provided exception handler (if one exists) at the end of a block statement or at the end of the body of a subprogram ♦, or package. If no handler exists, the exception is propagated out to the main program. 2

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. 2a

References: block statement 5.6, error situation 1.6, exception handler 11.2, name 4.1, package body 7.1, propagation of an exception 11.4.1 11.4.2, raise statement 11.3, subprogram body 6.3, 3

11.1 Exception Declarations

The following exceptions are predefined in the language; they are raised when the situations described are detected. 1

CONSTRAINT_ERROR 4

This exception is raised in any of the following situations: upon an attempt to violate a range constraint or an index constraint, by the execution of a predefined numeric operation that cannot deliver a correct result; this includes the case where an implementation uses a predefined numeric operation for the execution, evaluation, or elaboration of some construct [AI-00387].

NUMERIC_ERROR This exception is not raised. All previous occurences have been replaced by CONSTRAINT_ERROR [AI-00387]. 5

PROGRAM_ERROR 6

This exception is raised upon an attempt to call a subprogram ♦ if the body of the corresponding unit has not yet been elaborated or by an attempt to evaluate a

deferred constant before its full declaration has been elaborated (7.4.3). This exception is also raised if the end of a function is reached (see 6.5), by operations in the AVA_IO package, and may be explicitly raised by the programmer.

STORAGE_ERROR This exception may be raised in case available storage is not sufficient for the execution or elaboration of a construct [AI-00133]. 7

♦ 8

References: collection 3.8, declaration 3.1, exception 11, exception handler 11.2, identifier 2.3, implicit declaration 12.3, instantiation 12.3, name 4.1, object 3.2, raise statement 11.3, record component 3.7, return statement 5.8, subprogram 6, subprogram body 6.3. 12

Constraint_error exception contexts: aggregate 4.3.1 4.3.2, assignment statement 5.2 5.2.1, constraint 3.3.2, discrete type attribute 3.5.5, exponentiating operator 4.5.6, implicit conversion 3.5.4 3.5.6 4.6, index constraint 3.6.1, indexed component 4.1.1, logical operator 4.5.1, numeric operation 3.5.5 3.5.8 3.5.10, object declaration 3.2.1, operator of a numeric type 4.5 4.5.7, parameter association 6.4.1, qualified expression 4.7, range constraint 3.5, selected component, 4.1.3, subtype indication 3.3.2, type conversion 4.6. 13

Numeric_error exception contexts: Moved to constraint errror. 14

Program_error exception contexts: collection 3.8, elaboration 3.9, elaboration check 3.9 7.3 9.3 12.2, leaving a function 6.5 15

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
11.01	(03)	ra	WI	0334/01	87-02-12	Exceptions and recursive instantiation of generics
11.01	(06)	BI	WI	0159/00	86-08-01	NUMERIC_ERROR cannot be raised arbitrarily
11.01	(06)	BI	WJ	0311/06	86-07-23	No NUMERIC_ERROR for null strings
11.01	(06)	NB	WJ	0387/05	87-02-23	Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR
11.01	(06)	ra	RE	0115/00	83-11-07	NUMERIC_ERROR raised by implicit type conversion
11.01	(06)	ra	WJ	0312/04	86-07-23	NUMERIC_ERROR when evaluating null aggregates and slices
11.01	(08)	BI	RE	0133/03	86-07-10	raising STORAGE_ERROR
11.01	(08)	co	WI	0036/02	84-03-26	STORAGE_ERROR - null arrays; NUMERIC_ERROR - array types

11.2 Exception Handlers

The response to one or more exceptions is specified by an exception handler. 1

exception_handler ::=

when others =>
  sequence_of_statements 2

♦

An exception choice can only be **others**. An exception ♦ handler occurs in a construct that is either a block statement or the body of a subprogram ♦ or package. Such a construct will be called a *frame* in this chapter. In each case the syntax of a frame that has exception handlers includes the following part: 3

begin
  sequence_of_statements
exception 4

```
    exception_handler ♦
end
```

The exception choice **others** traps all exceptions. 5

The exception handler of a frame handles all exceptions that are raised by the execution of the sequence of statements of the frame. ♦ 6

Example: 7

```
begin
  -- sequence of statements
exception
  ♦
  when others =>
    PUT(standard_output," FATAL ERROR ");
    raise PROGRAM_ERROR;
end;
```

Note:

The same kinds of statement are allowed in the sequence of statements of an exception handler as are allowed in the sequence of statements of the frame. For example, a return statement is allowed in a handler within a function body. 8

References: block statement 5.6, declarative part 3.9, exception 11, exception handling 11.4, function body 6.3, name 4.1, package body 7.1, raise statement 11.3, return statement 5.8, sequence of statements 5.1, statement 5, subprogram body 6.3, visibility 8.3 9

11.3 Raise Statements

A raise statement raises an exception. 1

raise_statement ::= **raise** PROGRAM_ERROR; 2

For the execution of a raise statement, the exception PROGRAM_ERROR is raised. 3

Examples: 4

```
raise PROGRAM_ERROR;
```

References: exception 11, name 4.1, package 7, sequence of statements 5.1, subprogram 6 5

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
11.03	(02)	ST	RE	0450/00	86-08-06	Should allow raising of an exception in another task.

11.4 Exception Handling

When an exception is raised, normal program execution is abandoned and control is transferred to an exception handler. The selection of this handler depends on whether the exception is raised during the execution of statements or during the elaboration of declarations. 1

References: declaration 3.1, elaboration 3.1 3.9, exception 11, exception handler 11.2, raising of exceptions 11.3, statement 5 2

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
11.04	(00)	ST	RE	0595/00	88-10-05	Name of the "current exception"
11.04	(01)	BI	WJ	0446/05	87-09-25	Raising an exception in an abnormally completed task

11.4.1 Exceptions Raised During the Execution of Statements

The handling of an exception raised by the execution of a sequence of statements depends on ♦ the innermost frame that ♦ encloses the sequence of statements ♦. 1

Different actions take place, depending on whether or not this frame has a handler for the exception, and on whether the exception is raised in the sequence of statements of the frame or in that of an exception handler. 2

If an exception is raised in the sequence of statements of a frame that has a handler for the exception, execution of the sequence of statements of the frame is abandoned and control is transferred to the exception handler. The execution of the sequence of statements of the handler completes the execution of the frame (or its elaboration if the frame is a package body). 3

If an exception is raised in the sequence of statements of a frame that does not have a handler for the exception, execution of this sequence of statements is abandoned. The next action depends on the nature of the frame: 4

(a) For a subprogram body, the same exception is raised again at the point of call of the subprogram, unless the subprogram is the main program itself, in which case execution of the main program is abandoned. 5

(b) For a block statement, the same exception is raised again immediately after the block statement (that is, within the innermost enclosing frame). 6

(c) For a package body that is a declarative item, the same exception is raised again immediately after this declarative item (within the enclosing declarative part). ♦ If the package is a library unit, execution of the main program is abandoned. 7

(d) Omitted 8

An exception that is raised again (as in the above cases (a), (b), and (c)) is said to be *propagated*, either by the execution of the subprogram, the execution of the block statement, or the elaboration of the package body. ♦ 9

Finally, if an exception is raised in the sequence of statements of an exception handler, execution of this sequence of statements is abandoned. Subsequent actions (including propagation, if any) are as in the cases (a) to (d) above, depending on the nature of the frame. 10

Example: 11

```
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;
```

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. 12

It is poor programming style to 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. 12a

Example: 13

```
package P is
  procedure R;
  procedure Q;
end P;

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;

begin
  ...            -- error situation (1)
  Q;
  ...
exception
   when others =>     --  handler E1
  ...
end P;
```

The following situations can arise: 14

(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 elaboration of P.

(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. 15

(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). 16

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. 17

♦ 18

Notes:

The language does not define what happens when the execution of the main program is abandoned after an unhandled exception. 19

The predefined exceptions are those that can be propagated by the basic operations and the predefined operators. 20

♦ 21

References: basic operation 3.3.3, block statement 5.6, completion 9.4, declarative item 3.9, declarative part 3.9, elaboration 3.1 3.9, exception 11, exception handler 11.2, frame 11.2, library unit 10.1, main program 10.1, numeric_error exception 11.1, package 7, package body 7.1, predefined operator 4.5, procedure 6.1, sequence of statements 5.1, statement 5, subprogram 6, subprogram body 6.3, subprogram call 6.4 22

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
11.04.01	(03)	BI	WJ	0455/05	88-06-29	Raising an exception before the sequence of statements

11.4.2 Exceptions Raised During the Elaboration of Declarations

If an exception is raised during the elaboration of the declarative part of a given frame, this elaboration is abandoned. The next action depends on the nature of the frame: 1

(a) For a subprogram body, the same exception is raised again at the point of call of the subprogram, unless the subprogram is the main program itself, in which case execution of the main program is abandoned. 2

(b) For a block statement, the same exception is raised again immediately after the block statement. 3

(c) For a package body that is a declarative item, the same exception is raised again immediately after this declarative item, in the enclosing declarative part. ♦ If the package is a library unit, execution of the main program is abandoned. 4

(d) Omitted 5

Similarly, if an exception is raised during the elaboration of a package declaration ♦ this elaboration is 6

abandoned; the next action depends on the nature of the declaration.

(e) For a package declaration that is a declarative item, the exception is raised again immediately after the declarative item in the enclosing declarative part or package specification. For the declaration of a library package, the execution of the main program is abandoned. 7

An exception that is raised again (as in the above cases (a), (b), (c) and (e)) is said to be *propagated*, either by the execution of the subprogram or block statement, or by the elaboration of the package declaration, or package body. 8

Example of an exception in the declarative part of a block statement (case (b)): 9

```
procedure P is
  ...
begin
  declare
    N : INTEGER := F;     -- the function F may raise PROGRAM_ERROR
  begin                   -- enter frame body here
    ...
  exception
    when others => ...    -- handler E1
  end;
  ...
exception
  when others => ...      -- handler E2
end P;

-- if the exception PROGRAM_ERROR is raised in the declaration of N, it is handled by E2
```

References: activation 9.3, block statement 5.6, declarative item 3.9, declarative part 3.9, elaboration 3.1 3.9, exception 11, frame 11.2, library unit 10.1, main program 10.1, package body 7.1, package declaration 7.1, package specification 7.1, subprogram 6, subprogram body 6.3, subprogram call 6.4 10

11.5 Exceptions Raised During Task Communication: Removed

11.6 Exceptions and Optimization: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
11.06	(00)	BI	RE	0315/11	87-04-16	Legal Reorderings of Operations
11.06	(05)	ra	RE	0380/00	85-08-22	Reassociation and overloading resolution
11.06	(06)	ra	WJ	0267/06	88-07-07	Evaluating expressions in case statements

11.7 Suppressing Checks: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
11.07	(03)	ra	RE	0299/00	84-10-16	Pragma Suppress and Subprogram Names
11.07	(04)	BI	RE	0542/00	87-08-05	Meaning of "base type" wrt pragma SUPPRESS
11.07	(10)	BI	WA	0535/03	88-11-04	Boolean operators producing out of range results
11.07	(18)	ra	RE	0532/00	87-03-11	Pragma SUPPRESS and compile-time evaluation of expressions

Chapter 12

GENERIC UNITS: REMOVED

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
12.01	(02)	ST	RE	0382/00	85-08-22	Allow generic subprogram bodies
12.01	(02)	ST	RE	0451/00	86-08-06	Task entries as formal parameters to generics
12.01	(02)	ST	RE	0452/00	86-08-06	"generic_type_definition" should have generic record types
12.01	(05)	BI	WA	0504/03	88-06-16	Expanded names with a renamed prefix in generic packages
12.01	(05)	BI	WJ	0286/11	87-06-18	Declarations visible in a generic subprogram decl and body
12.01	(05)	BI	WJ	0367/06	88-05-23	Deriving from types declared in a generic package
12.01	(05)	BI	WJ	0412/06	88-05-23	Expanded names for generic formal parameters
12.01	(05)	co	RE	0335/00	85-05-02	An expanded name of a generic is also "the name"
12.02	(01)	ra	WJ	0328/08	87-03-16	Legality of uninstantiated generic units
12.03	(00)	BI	CB	0506/04	88-11-08	Instantiation before occurrence of a body
12.03	(05)	BI	CA	0547/03	88-10-05	Conformance rules and instantiated units
12.03	(05)	BI	WJ	0398/08	87-06-18	Operations declared for types declared in instances
12.03	(05)	BI	WJ	0409/05	87-09-12	Static subtype names created by instantiation
12.03	(05)	CR	WJ	0483/04	87-09-12	Correction to question in 0409/03
12.03	(05)	na	na	0290/02	86-07-31	[combined with 0398]
12.03	(07)	BI	WA	0505/03	88-06-16	Static constants in instances
12.03	(17)	BI	WJ	0237/06	86-07-23	Instances having implicit package bodies
12.03	(17)	BI	WJ	0365/05	86-07-23	Actual parameter names are evaluated in generic instantiations
12.03	(18)	BI	WA	0256/23	88-06-16	"Successful" compilation
12.03	(22)	BI	WJ	0012/06	88-05-23	Declaration of homographs by derivation and instantiation
12.03.02	(04)	BI	WA	0256/23	88-06-16	"Successful" compilation
12.03.02	(04)	BI	WJ	0037/12	86-12-01	Instantiating when discriminants have defaults
12.03.06	(02)	ra	WJ	0038/06	87-02-23	Declarations associated with default names

Chapter 13
REPRESENTATION CLAUSES AND IMPLEMENTATION-DEPENDENT FEATURES

This chapter describes certain implementation-dependent features ♦. 1

13.1 Representation Clauses: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.01	(03)	BI	WJ	0138/10	87-02-23	Representation clauses for derived types
13.01	(03)	BI	WJ	0422/06	88-05-23	Representation clauses for derived enumeration and record types
13.01	(03)	ra	RE	0514/00	87-01-13	Do representation clauses affect prior subtypes?
13.01	(03)	ra	WJ	0040/07	86-12-01	Multiple specification of T'SIZE, T'STORAGE_SIZE, T'SMALL
13.01	(06)	BI	WI	0494/02	87-01-16	Forcing occurrences for component types
13.01	(06)	BI	WI	0515/01	87-01-19	Expressions, function calls, and forcing occurrences
13.01	(06)	BI	WJ	0039/12	86-07-23	Forcing occurrences and premature uses of a type
13.01	(06)	BI	WJ	0186/08	86-12-04	Pragmas recognized by an impl do not force default representation
13.01	(06)	BI	WJ	0321/02	86-07-23	Forcing occurrence of index subtype
13.01	(06)	BI	WJ	0322/02	86-07-23	Forcing occurrences in unknown pragmas
13.01	(06)	na	na	0171/01	84-11-05	[combined with 0039]
13.01	(06)	ra	CE	0419/02	87-01-28	Forcing occurrence in a generic formal type declaration
13.01	(07)	BI	WI	0515/01	87-01-19	Expressions, function calls, and forcing occurrences
13.01	(07)	BI	WJ	0039/12	86-07-23	Forcing occurrences and premature uses of a type
13.01	(07)	BI	WJ	0371/05	86-07-23	Representation clauses containing forcing occurrences
13.01	(08)	na	na	0423/01	87-01-20	[combined with 0411]
13.01	(08)	ra	WI	0424/01	87-08-07	Storage size specification vs. address clause
13.01	(10)	BI	CA	0523/05	88-10-03	Changing the effect of a program by specifying *small*
13.01	(10)	BI	RE	0383/00	85-08-22	Need for "erroneous" address clauses

13.2 Length Clauses: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.02	(02)	BI	WJ	0300/07	87-06-18	Prefixes of attributes in length clauses
13.02	(03)	co	RE	0436/00	86-06-19	A static expression must not be evaluated?
13.02	(05)	BI	WI	0553/01	88-11-22	Giving a size specification for a record type
13.02	(05)	BI	WI	0561/02	88-12-13	Size specifications for scalar types
13.02	(05)	ra	CE	0369/06	88-11-20	Representing values of discrete base type
13.02	(06)	na	na	0495/02	88-05-10	[Combined with 0301]
13.02	(06)	ra	WI	0301/03	88-10-11	Representation clauses and nonstatic constraints
13.02	(08)	ra	RE	0496/00	86-11-10	Specifying a negative STORAGE_SIZE value for tasks

13.02	(10)	BI	CE	0555/02	88-11-22	Pragma PACK for an array type
13.02	(10)	BI	WI	0554/01	88-11-22	Pragma PACK for a record type
13.02	(10)	BI	WI	0556/03	88-11-22	Giving a size specification for an array type
13.02	(10)	ST	RE	0453/00	86-08-06	STORAGE_SIZE for Tasks
13.02	(12)	BI	CE	0341/13	88-12-07	Extra precision or range for fixed point representations
13.02	(12)	ra	WJ	0099/12	88-05-23	'SMALL can be specified for a derived fixed point type
13.02	(14)	BI	RE	0497/00	86-11-10	STORAGE_SIZE and dependent tasks
13.02	(17)	ra	RE	0524/00	87-02-10	Spanning the Range of the Type

13.3 Enumeration Representation Clauses: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.03	(00)	BI	WI	0564/00	88-10-11	Required enumeration representation clause support
13.03	(02)	BI	WJ	0422/06	88-05-23	Representation clauses for derived enumeration and record types
13.03	(04)	BI	RE	0798/00	88-12-21	Replace "values" by "expressions"

13.4 Record Representation Clauses: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.04	(02)	BI	WJ	0422/06	88-05-23	Representation clauses for derived enumeration and record types
13.04	(03)	BI	RE	0498/00	86-11-10	static range in component_clause
13.04	(05)	BI	CE	0550/03	88-11-22	Positioning record components across storage unit boundaries
13.04	(05)	BI	WI	0551/04	88-12-13	Specifying a record component size in a component clause
13.04	(07)	BI	CE	0550/03	88-11-22	Positioning record components across storage unit boundaries
13.04	(07)	BI	WI	0551/04	88-12-13	Specifying a record component size in a component clause
13.04	(07)	BI	WJ	0132/05	86-07-23	Static constraints and component clauses
13.04	(07)	ra	RE	0499/00	86-11-10	what is a record variant?
13.04	(07)	ra	WI	0301/03	88-10-11	Representation clauses and nonstatic constraints
13.04	(08)	BI	CE	0009/04	88-09-02	Implementation-defined names cannot be reserved words
13.04	(08)	BI	CE	0552/02	88-11-22	Component clauses for implementation-dependent components

13.5 Address Clauses: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.05	(00)	BI	RE	0573/01	88-11-08	Address clause and Initial values
13.05	(00)	co	na	0338/02	85-12-30	An address clause is illegal after a forcing occurrence [13.1(8)]
13.05	(00)	na	na	0228/01	87-08-03	[combined with 0337]
13.05	(03)	BI	WI	0559/00	88-10-07	Support for address clauses for objects and program units
13.05	(03)	ra	WI	0337/04	88-09-02	Incompatible record alignment and address clauses
13.05	(04)	BI	WJ	0263/06	88-05-23	A named number is not an object
13.05	(05)	ra	WJ	0336/05	88-05-23	Address clauses for subprogram bodies
13.05	(08)	BI	WJ	0292/05	86-12-01	Derived types with address clauses for entries
13.05	(08)	co	WJ	0379/03	86-12-01	Address clauses for entries of task types

13.5.1 Interrupts: Removed

13.6 Change of Representation: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.06	(01)	BI	WJ	0138/10	87-02-23	Representation clauses for derived types
13.06	(01)	ra	WJ	0040/07	86-12-01	Multiple specification of T'SIZE, T'STORAGE_SIZE, T'SMALL

13.7 The Package AVA

For each implementation there is a predefined library package called AVA[69] which includes the definitions of certain configuration-dependent characteristics. The specification of the package AVA is implementation-dependent and must be given in Appendix F. The visible part of this package must contain at least the following declarations. 1

```
package AVA is                                                    2
   ♦
   type NAME is implementation_defined_enumeration_type;

   AVA_SYSTEM_NAME : constant NAME := implementation_defined;

   ♦

   -- System-Dependent Named Numbers:

   AVA_MIN_INT  : constant := implementation_defined;
   AVA_MAX_INT  : constant := implementation_defined;
   ♦

end AVA;
```

♦ 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. ♦ 3

Note:

It is a consequence of the visibility rules that a declaration given in the package AVA 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. 12

References: apply 10.1.1, attribute 4.1.4, compilation unit 10.1, declaration 3.1, enumeration literal 3.5.1, enumeration type 3.5.1, identifier 2.3, library unit 10.1, must 1.6, named number 3.2, number declaration 3.2.2, numeric literal 2.4, package 7, package specification 7.1, program library 10.1, type 3.3, visibility 8.3, visible part 7.2, with clause 10.1.1 13

[69]The package AVA replaces the Ada package SYSTEM.

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.07	(00)	BI	WI	0041/05	87-04-19	Recompilation of predefined library units
13.07	(02)	co	WJ	0045/05	86-07-23	Subtype SYSTEM.PRIORITY
13.07	(02)	ra	WJ	0355/06	86-12-01	Pragma ELABORATE for predefined library packages
13.07	(02)	ST	RE	0582/00	88-08-31	Need a standard name for null address
13.07	(03)	ST	RE	0291/00	84-10-01	SYSTEM.MAX_DIGITS Insufficient for Portability
13.07	(06)	BI	RE	0229/00	84-03-13	Enumeration literals include characters
13.07	(11)	BI	WI	0044/02	87-04-08	Pragma SYSTEM_NAME in nonempty library
13.07	(11)	BI	WI	0303/01	87-04-14	Implicit use of SYSTEM.STORAGE_UNIT
13.07	(11)	ra	WI	0302/01	87-04-10	Pragma SYSTEM_NAME (and related pragmas)

13.7.1 System-Dependent Named Numbers

Within the package AVA, the following named numbers are declared. All are of the type *universal_integer*. ♦ 1

AVA_MIN_INT	The smallest (most negative) value of INTEGER type. 2
AVA_MAX_INT	The largest (most positive) value of INTEGER type. 3

♦ 4

References: allow 1.6, integer type 3.5.4, named number 3.2, package 7, range constraint 3.5, system package 13.7, type 3.3, universal_integer type 3.5.4 8

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.07.01	(02)	BI	CA	0565/02	88-10-03	Support for static *universal_integer* expressions
13.07.01	(02)	BI	WA	0304/05	88-11-04	The definition of SYSTEM.MIN_INT and SYSTEM.MAX_INT
13.07.01	(03)	BI	CA	0565/02	88-10-03	Support for static *universal_integer* expressions
13.07.01	(03)	BI	WA	0304/05	88-11-04	The definition of SYSTEM.MIN_INT and SYSTEM.MAX_INT
13.07.01	(07)	ra	WJ	0201/07	88-05-23	The relation between TICK, CLOCK, and the delay statement
13.07.01	(07)	ra	WJ	0366/07	88-05-23	The value of SYSTEM.TICK for different execution environments

13.7.2 Representation Attributes: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.07.02	(03)	BI	CE	0043/03	88-05-09	SYSTEM must apply to a unit using ’ADDRESS
13.07.02	(03)	BI	RE	0596/00	88-10-05	’ADDRESS for derived task types
13.07.02	(03)	BI	RE	0603/00	88-11-08	ADDRESS attribute for single tasks
13.07.02	(03)	BI	WJ	0305/05	88-05-23	T’ADDRESS when T is a task type yields the task object address
13.07.02	(03)	ra	RE	0202/00	84-03-13	’ADDRESS may not be Finer-grained than STORAGE_UNIT
13.07.02	(03)	ra	RE	0203/02	86-06-19	How is ’ADDRESS defined?
13.07.02	(03)	ra	RE	0533/00	87-03-11	Interpretation of ’address for a task entry
13.07.02	(05)	BI	RE	0536/03	88-11-08	The meaning of ’SIZE applied to a type or object
13.07.02	(06)	ra	na	0126/11	84-09-28	[combined with 0015]
13.07.02	(06)	ra	WJ	0015/12	86-12-01	When the prefix of ’ADDRESS contains a function name
13.07.02	(07)	BI	WJ	0258/06	88-06-13	’POSITION etc. for renamed components
13.07.02	(07)	CR	WJ	0503/04	88-05-23	Error in 0258/05
13.07.02	(08)	co	WJ	0362/03	86-12-01	"component of a record" for representation attributes
13.07.02	(12)	BI	RE	0608/00	88-12-13	The value of ’STORAGE_SIZE

13.7.3 Representation Attributes of Real Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.07.03	(03)	BI	WJ	0263/06	88-05-23	A named number is not an object
13.07.03	(03)	ra	WI	0238/02	87-03-11	Effect of MACHINE_ROUNDS for underflow
13.07.03	(04)	BI	CA	0174/07	88-10-03	T'FIRST and T'LAST for real types
13.07.03	(04)	BI	WI	0021/12	88-11-04	MACHINE_OVERFLOWS for correct extended safe results
13.07.03	(05)	BI	WJ	0263/06	88-05-23	A named number is not an object
13.07.03	(05)	ST	RE	0609/00	88-12-13	Floating point machine attributes inadequate
13.07.03	(09)	co	CE	0543/01	88-12-08	The value of F'MACHINE_RADIX exp() (F'MACHINE_EMIN - 1)

13.8 Machine Code Insertions: Removed

13.9 Interface to Other Languages: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.09	(00)	na	na	0326/01	86-03-25	[combined with 0306]
13.09	(03)	BI	RE	0805/00	88-12-21	"{basic} declarative item" (twice)
13.09	(03)	BI	WJ	0180/07	86-07-23	Elaboration checks for INTERFACE subprograms
13.09	(03)	BI	WJ	0306/15	88-05-23	Pragma INTERFACE: allowed names and illegalities
13.09	(03)	co	RE	0454/00	86-08-06	Allowed placement of pragma INTERFACE
13.09	(03)	na	na	0410/02	86-07-01	[Combined with 0306]
13.09	(03)	ra	WJ	0298/05	86-12-01	Interaction between pragmas ELABORATE and INTERFACE
13.09	(07)	na	na	0372/01	86-03-25	[combined with 0317]

13.10 Unchecked Programming: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
13.10.01	(01)	ra	WJ	0355/06	86-12-01	Pragma ELABORATE for predefined library packages
13.10.01	(06)	BI	WJ	0356/08	88-05-23	Access values that designate deallocated objects
13.10.01	(08)	ra	RE	0589/00	88-09-02	Insert notes to clarify the use of UNCHECKED_DEALLOCATION
13.10.02	(00)	BI	RE	0500/00	86-11-10	UNCHECKED_CONVERSION for unconstrained types
13.10.02	(01)	ra	WJ	0355/06	86-12-01	Pragma ELABORATE for predefined library packages
13.10.02	(03)	ra	RE	0590/00	88-09-02	Safe unchecked conversions

Chapter 14
INPUT-OUTPUT

Input-output is provided in the language by means of predefined packages. ♦ Operations for text input-output are supplied in the package AVA_IO.[70] ♦ 1

References: ♦ AVA_IO package 14.3 2

14.1 External Files and File Objects

Values input from the external environment of the program, or output to the 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 (see Appendix F). External files are made available via parameters to the main program of type *file type* and by deferred constants declared in the package AVA_IO. These constants must include at least STANDARD_INPUT and STANDARD_OUTPUT. Any aliasing among external file parameters, standard_input and standard_output will result in implementation dependent behavior [AI-00320]. 1

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 chapter,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. ♦ 2

♦ 3

♦ Input-output in human-readable form is defined by the (nongeneric) package AVA_IO. 5

Before input or output operations can be performed on a file, the file must first 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

---

[70]Note that this package should be trivially implementable using Ada's predefined TEXT_IO. The reason we define a completely new package is in order to avoid turning ambiguous Ada programs into unambiguious AVA programs. TEXT_IO makes extensive use of default parameters, which we have excluded from our subset.

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 enumeration type 8

```
♦
type FILE_MODE is (IN_FILE, OUT_FILE);          -- for AVA_IO
```

These values correspond respectively to the cases where only reading ♦ or only writing are to be performed. The mode of a file cannot be changed. 9

♦ 10

The only exception that can be raised by a call of an input-output subprogram is PROGRAM_ERROR;[71] the situations in which it can be raised are described, either following the description of the subprogram (and in section 14.4), or in Appendix F in the case of error situations that are implementation-dependent. 11

Notes:

Other exceptions may be raised by the evaluation of the *arguments* to input-output subprograms, but after execution of the body has begun the only exceptions that may be propagated to the caller is PROGRAM_ERROR [AI-00279]. 12

♦ 13

References: ♦ exception 11, file mode 14.1, ♦ io_exceptions package 14.5, open file 14.1, ♦ output file 14.2.2, ♦ string 3.6.3, AVA_IO package 14.3 ♦ 14

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.01	(00)	BI	RE	0591/00	88-09-02	Concurrent I/O and erroneous executions
14.01	(00)	BI	RE	0592/00	88-09-02	I/O with undefined parameters
14.01	(01)	ra	WJ	0355/06	86-12-01	Pragma ELABORATE for predefined library packages
14.01	(07)	BI	WJ	0466/04	88-05-23	I/O performed by library tasks
14.01	(11)	BI	WJ	0279/09	86-12-01	Exceptions raised by calls of I/O subprograms
14.01	(11)	ra	CE	0501/02	88-12-30	The open/closed status of a file object

14.2 Sequential and Direct Files

One kind of access to external files is defined: *sequential access*♦. ♦ A file object to be used for sequential access is called a *sequential file*. ♦ 1

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 environment). When the file is opened, transfer starts from the beginning of the file. 2

[71]To help the reader relate these exceptions to those defined for TEXT_IO in Ada, we use the notation $PROGRAM\_ERROR_{original}$ to indicate what original exception was replaced by program_error. E.g. $PROGRAM\_ERROR_{status}$, $PROGRAM\_ERROR_{mode}$, ...

♦

♦The only allowed modes for sequential files are the modes IN_FILE and OUT_FILE. 5

Notes:

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. 5a

References: count type 14.3, file mode 14.1, in_file 14.1, out_file 14.1 6

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.02	(02)	ra	CA	0278/05	88-10-03	Appending to a file

14.2.1 File Management

Relevant portions moved to 14.3.1.

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.02.01	(00)	ST	RE	0544/00	87-08-05	File "append" capability proposed
14.02.01	(00)	ST	RE	0545/00	87-08-05	Procedure to find if a file exists
14.02.01	(03)	co	WJ	0247/05	86-07-23	A non-null FORM argument can be required by an implementation
14.02.01	(03)	na	na	0484/00	86-10-15	Correction to 0048/06 example
14.02.01	(03)	ra	WJ	0046/06	86-07-23	Lifetime of a temporary file and its name
14.02.01	(04)	BI	WJ	0332/04	86-07-23	NAME_ERROR or USE_ERROR raised when I/O not supported
14.02.01	(06)	ra	RE	0574/00	88-07-06	Effect of invalid FORM string
14.02.01	(07)	BI	WJ	0332/04	86-07-23	NAME_ERROR or USE_ERROR raised when I/O not supported
14.02.01	(09)	BI	WJ	0357/05	87-06-18	CLOSE or RESET of a sequential file from OUT_FILE mode
14.02.01	(15)	BI	WJ	0357/05	87-06-18	CLOSE or RESET of a sequential file from OUT_FILE mode
14.02.01	(20)	ra	RE	0448/00	86-07-10	Semantic ramifications of the NAME interface
14.02.01	(22)	ra	WJ	0046/06	86-07-23	Lifetime of a temporary file and its name

14.2.2 Sequential Input-Output

Relevant portions moved to 14.3.1.

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.02.02	(00)	BI	WJ	0320/06	86-07-23	Sharing external files
14.02.02	(04)	ra	RE	0184/00	84-01-25	DATA_ERROR in SEQUENTIAL_IO and DIRECT_IO

14.2.3 Specification of the Package Sequential_IO: Removed

14.2.4 Direct Input-Output: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.02.04	(00)	BI	WJ	0320/06	86-07-23	Sharing external files

14.2.5 Specification of the Package Direct_IO: Removed

14.3 Text Input-Output

This section describes the package AVA_IO, which provides facilities for input and output in human-readable form. Each file value is read or written sequentially, as a sequence of characters ♦. The specification of the package is given below in section 14.3.10. 1

The facilities for file management given ♦ in section ♦ 14.3.1, are available for text input-output. ♦ There are 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

♦ 3

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

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. This terminator is generated during output ♦ by calls of procedures provided expressly for that purpose; or by passing the values of this constant as a character to be output.[72] 6

♦ 7

♦ When a file is initially open with mode OUT_FILE, its size is unbounded. STORAGE_ERROR is raised if external file size limits are encountered. ♦ 9

References: ♦ external file 14.1, file 14.1, get procedure 14.3.5, in_file 14.1, out_file 14.1, put procedure 14.3.5, ♦ sequential access 14.1, standard input file 14.3.2, standard output file 14.3.2 10

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03	(00)	ST	RE	0329/00	85-03-05	look-ahead operation for TEXT_IO
14.03	(01)	ra	WJ	0355/06	86-12-01	Pragma ELABORATE for predefined library packages
14.03	(05)	ra	RE	0575/00	88-07-06	Exception handling when Standard I/O is not supported
14.03	(05)	ST	RE	0485/00	86-10-13	Must standard input and output files be independent?
14.03	(06)	ra	RE	0537/00	87-06-11	Raising END_ERROR on SKIP_PAGE

14.3.1 File Management

The only allowed file modes for text files are the modes IN_FILE and OUT_FILE. The subprograms IS_OPEN, MODE, and CLOSE are described in this section.[73] 1

```
function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;
```
1a

[72] This means that an entire file can be copied by GETting and PUTting characters, without any knowledge of the underlying file structure.

[73] These have been moved here from 14.2.1.

Returns TRUE if FILE is open. 1b

function MODE (FILE : **in** FILE_TYPE) **return** FILE_MODE; 1c

Returns the FILE_MODE of FILE, either IN_FILE or OUT_FILE. 1d

procedure CLOSE (FILE : **in** FILE_TYPE); 1e

Severs the association between the given file and its associated external file. The given file is left closed. 1f

If the file has the current mode OUT_FILE♦ outputs a file terminator. 1g

The exception PROGRAM_ERROR$_{status}$ is raised if the given file is not open. 1h

function END_OF_FILE[74](FILE : **in** FILE_TYPE) **return** BOOLEAN; 1i

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 PROGRAM_ERROR$_{status}$ is raised if this operation is attempted for a file that is not open. 1j

The exception PROGRAM_ERROR$_{mode}$ is raised if the mode is not IN_FILE. 1k

♦ 2

References: current mode 14.1, current size 14.1, closed file 14.1, ♦ current column number 14.3, current default input file 14.3, current line number 14.3, current page number 14.3, end_of_file 14.3, program_error exception 14.4, external file 14.1, file 14.1, file mode 14.1, file terminator 14.3, in_file 14.1, line length 14.3, ♦ out_file 14.1, page length 14.3, ♦ 6

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.01	(02)	ra	CA	0278/05	88-10-03	Appending to a file
14.03.01	(04)	BI	WJ	0047/08	87-09-12	Effect of RESET on line and page length
14.03.01	(04)	CR	WJ	0486/04	87-09-12	Correction to 0047/06 example
14.03.01	(05)	BI	WJ	0048/12	87-02-23	Default files can be closed, deleted, and re-opened

14.3.2 Default Input and Output Files

♦ 1

The following constants provide one means to access the file pointers to the standard input and output files. 1a

STANDARD_INPUT : **constant** FILE_TYPE; 1b

Value is a pointer to the standard input file (see 14.3). 1c

STANDARD_OUTPUT : **constant** FILE_TYPE; 1d

Value is a pointer to the standard output file (see 14.3). 1e

♦

[74]Moved from 14.2.2.

References: ♦ file_type 14.1, get procedure 14.3.5, mode_error exception 14.4, put procedure 14.3.5, PROGRAM_ERROR$_{status}$ exception 14.4 6

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.02	(00)	ra	WI	0049/01	84-01-29	Operations on an out-of-scope file object
14.03.02	(01)	BI	RE	0546/00	87-08-05	CREATE vis-a-vis 0048
14.03.02	(01)	BI	WJ	0048/12	87-02-23	Default files can be closed, deleted, and re-opened

14.3.3 Specification of Line and Page Lengths: Omitted

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.03	(05)	ra	CE	0534/03	88-10-03	Unbounded line lengths

14.3.4 Operations on Columns, Lines, and Pages

♦ 1

The end of line constant described in this section and the procedures PUT_LINE and GET_LINE described in section 14.3.6 provide the only explicit control of line structure in files. 1a

EOL : **constant** CHARACTER := *implementation_dependent*; 1b

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.04	(00)	BI	WJ	0320/06	86-07-23	Sharing external files
14.03.04	(21)	ST	RE	0487/00	86-10-13	The TEXT_IO procedures end_of_page and end_of_file

14.3.5 Get and Put Procedures

The procedures GET and PUT for items of the types CHARACTER and STRING ♦ are described in subsequent sections. Features of these procedures that are common to ♦ these types are described in this section. The GET and PUT procedures for items of type CHARACTER and STRING deal with sequences of individual character values rather than Ada literals. 1

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. 2

♦ 3

The exception PROGRAM_ERROR$_{status}$ is raised 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 raised 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. 9

The exception PROGRAM_ERROR$_{end}$ is raised by a GET procedure if an attempt is made to read past the end of a file.♦ 10

♦ 11

References: blank 14.3.9, column number 14.3, current default file 14.3, ♦ file 14.1, ♦ get procedure 14.3.6 14.3.7 14.3.8 14.3.9, in_file 14.1, ♦ line number 14.1, line terminator 14.1, maximum line length 14.3, mode 14.1, ♦ new_file procedure 14.3.4, out_file 14.1, page number 14.1, page terminator 14.1, put procedure 14.3.6 14.3.7 14.3.8 14.3.9, skipping 14.3.7 14.3.8 14.3.9, ♦ 14

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.05	(03)	BI	WJ	0320/06	86-07-23	Sharing external files
14.03.05	(07)	co	WJ	0243/05	86-12-01	Overriding width format in TEXT_IO

14.3.6 Input-Output of Characters and Strings

For an item of type CHARACTER the following procedures are provided: 1

procedure GET(FILE : **in** FILE_TYPE; ITEM : **in out**[75] CHARACTER); 2
♦

♦ Reads the next character from the specified input file and returns the value of this character in the **out** parameter ITEM. 3

The exception PROGRAM_ERROR$_{end}$ is raised if an attempt is made to read past the end of a file. 4

procedure PUT(FILE : **in** FILE_TYPE; ITEM : **in** CHARACTER); 5
♦

♦ Outputs the given character to the file. 6

For an item of type STRING the following procedures are provided: 7

procedure GET(FILE : **in** FILE_TYPE; ITEM : **in out** STRING); 8
♦

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). 9

procedure PUT(FILE : **in** FILE_TYPE; ITEM : **in** STRING); 10
♦

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). 11

procedure GET_LINE(FILE : **in** FILE_TYPE; ITEM : **in out**[76] STRING; LAST : **in out** NATURAL); 12
♦

Replaces successive characters of the specified string by successive characters read from the specified input file. Reading stops if ♦ the end of the string is met. Reading stops if EOL is met or if the end of the file is encountered. If an EOL ended the GET_LINE, it is skipped. Characters not replaced in the string are left unchanged. 13

If characters are read, returns in LAST the index value such that ITEM(LAST) is the last character replaced (the index of the first character replaced is 0). If no characters are read, returns in LAST an index value that is one less than ITEM'FIRST 0. 14

---

[75]Because of the constraints that AVA places on parameter modes, excluding mode **out**, this parameter is **in out** rather than **out**.

[76]See previous footnote.

The exception PROGRAM_ERROR$_{end}$ is raised if an attempt is made to read past the end of a file. 15

procedure PUT_LINE(FILE : **in** FILE_TYPE; ITEM : **in** STRING); 16
♦

Calls the procedure PUT for the given string, and then ♦ outputs an EOL. 17

Notes:

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). 18

End of lines encountered by GET will be skipped over, while PUT may insert a number of them in the course of outputting the string. 19

References: current column number 14.3, ♦ file 14.1, file terminator 14.3, get procedure 14.3.5, line 14.3, line length 14.3, new_line procedure 14.3.4, page terminator 14.3, put procedure 14.3.4, skipping 14.3.5 20

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.06	(00)	ST	RE	0605/00	88-11-22	Can’t correctly read a file written with Text_IO
14.03.06	(03)	ST	RE	0488/00	86-10-13	Skipping of leading line terminators in get routines
14.03.06	(13)	BI	WJ	0050/11	86-12-01	When does GET_LINE call SKIP_LINE?
14.03.06	(13)	ra	WJ	0172/06	86-07-23	GET_LINE for interactive devices

14.3.7 Input-Output for Integer Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.07	(06)	BI	WJ	0051/07	86-07-23	Reading "integer literals"
14.03.07	(06)	ra	WJ	0307/04	86-07-23	GET at end of file and from a null string
14.03.07	(14)	ra	WJ	0307/04	86-07-23	GET at end of file and from a null string

14.3.8 Input-Output for Real Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.08	(09)	ra	WJ	0307/04	86-07-23	GET at end of file and from a null string
14.03.08	(18)	ra	WJ	0307/04	86-07-23	GET at end of file and from a null string
14.03.08	(20)	BI	WJ	0215/05	86-07-23	Type of EXP should be FIELD

14.3.9 Input-Output for Enumeration Types: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.09	(06)	NB	WJ	0239/11	87-02-23	ENUMERATION_IO and IMAGE for non-graphic characters
14.03.09	(06)	ra	WJ	0307/04	86-07-23	GET at end of file and from a null string
14.03.09	(06)	ra	WJ	0316/05	87-06-18	Definition of blank, inclusion of horizontal tab
14.03.09	(09)	NB	WJ	0239/11	87-02-23	ENUMERATION_IO and IMAGE for non-graphic characters
14.03.09	(11)	ra	WJ	0307/04	86-07-23	GET at end of file and from a null string

14.3.10 Specification of the Package AVA_IO

```
package AVA_IO is                                                    1

   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;
   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;
```

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.03.10	(00)	BI	RE	0248/00	84-05-14	Adding a private part to INTEGER_IO, etc.

14.4 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 F. ♦ 1

The exception $\text{PROGRAM\_ERROR}_{\text{status}}$ is raised by an attempt to read or write a file that is not open ♦. 2

The exception $\text{PROGRAM\_ERROR}_{\text{mode}}$ is raised 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. ♦ 3

♦ 4

The exception $\text{PROGRAM\_ERROR}_{\text{device}}$ is raised if an input-output operation cannot be completed because of a malfunction of the underlying system. 5

The exception $\text{PROGRAM\_ERROR}_{\text{use}}$ is raised if an operation is attempted that is not possible for reasons that depend on characteristics of the external file. ♦ 6

The exception $\text{PROGRAM\_ERROR}_{\text{end}}$ is raised by an attempt to skip (read past) the end of a file. 7

♦ 8

References: col function 14.3.4, ♦ end_of_line function 14.3.4, end_of_page function 14.3.4, external file 14.1, file 14.1, ♦ get procedure 14.3.5, in_file 14.1, io_exceptions package 14.5, line function 14.3.4, line_length function 14.3.4, name string 14.1, new_line procedure 14.3.4, new_page procedure 14.3.4, ♦ out_file 14.1, page function 14.3.4, page_length function 14.3.4, put procedure 14.3.5, ♦ skip_line procedure 14.3.4, skip_page procedure 14.3.4, AVA_IO package 14.3 10

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.04	(01)	BI	RE	0604/00	88-11-08	Precedence of DEVICE_ERROR and USE_ERROR
14.04	(04)	BI	WJ	0332/04	86-07-23	NAME_ERROR or USE_ERROR raised when I/O not supported
14.04	(05)	BI	WJ	0332/04	86-07-23	NAME_ERROR or USE_ERROR raised when I/O not supported
14.04	(05)	ra	RE	0576/00	88-07-06	I/O to strings when TEXT_IO is not supported
14.04	(08)	ra	RE	0577/00	88-07-06	Input position after raising DATA_ERROR

14.5 Specification of the Package IO_Exceptions: Removed

14.6 Low Level Input-Output: Removed

AI Crossreferences:

Section		Class	Status	AI-0	Date	Description
14.06	(05)	ra	WJ	0355/06	86-12-01	Pragma ELABORATE for predefined library packages
14.06	(05)	ST	WI	0003/01	84-03-26	Allow DATA of mode "in" in SEND_CONTROL

14.7 Example of Input-Output (Rewritten Example)

The following example shows the use of some of the text input-output facilities in a dialogue with a user at a terminal. The user is prompted to type a color, and the program responds by giving the number of items of that color available in stock, according to an inventory. The default input and output files are used. ♦ 1

```
with AVA_IO; use AVA_IO;                                                    2
package EXAMPLE is

  type COLOR is (WHITE, RED, ORANGE, YELLOW, GREEN, BLUE, BROWN);
  subtype color_range is integer range 1..7 ;

  type array_of_color_inventory is array (COLOR_RANGE) of INTEGER ;

  procedure DIALOGUE ;
  procedure ENTER_COLOR (SELECTION : in out COLOR_range) ;

end EXAMPLE ;

package body EXAMPLE is

  procedure ENTER_COLOR (SELECTION : in out COLOR_range) is
    NAME : string(1..8) := ".......";
    LENGTH : INTEGER := 0;
  begin
    loop
      PUT_LINE(standard_output,"Color selected: ");  -- prompts user
      GET_LINE(standard_input,NAME,LENGTH);      --  accepts color typed, or raises exception
      if    NAME = "white..."                    then SELECTION := 1; exit;
      elsif NAME = "red....."                    then SELECTION := 2; exit;
      elsif NAME = "orange.."                    then SELECTION := 3; exit;
      elsif NAME = "yellow.."                    then SELECTION := 4; exit;
      elsif NAME = "green..."                    then SELECTION := 5; exit;
      elsif NAME = "blue...."                    then SELECTION := 6; exit;
      elsif NAME = "brown..."                    then SELECTION := 7; exit;
      else  PUT_LINE(standard_output,"Invalid color, try again.  ");
            PUT(standard_output,EOL);
      end if;
      NAME := ".......";
    end loop;  --  repeats the loop until color accepted
  end;

  procedure DIALOGUE is
```

```
    INVENTORY : ARRAY_OF_COLOR_INVENTORY := (20, 17, 43, 10, 28, 173, 87);
    CHOICE : COLOR_RANGE := 1;
    begin
      loop

        ENTER_COLOR(CHOICE);  --  user types color and new line

        PUT(standard_output,"     ");
        PUT(standard_output,COLOR'IMAGE(COLOR'VAL(CHOICE)));
        PUT(standard_output," items available:");
        PUT(standard_output,"     ");
        PUT_LINE(standard_output,INTEGER'IMAGE(INVENTORY(CHOICE)));

      end loop;
    end DIALOGUE;

  end EXAMPLE;
```

Example of an interaction (characters typed by the user are italicized): 2a

```
Color selected:  Black
Invalid color, try again.

Color selected:  Blue
    BLUE items available:     173
Color selected:  Yellow
    YELLOW items available:     10
```

Appendix A
Predefined Language Attributes

This annex summarizes the definitions given elsewhere of the predefined language attributes.

♦

P'BASE — For a prefix P that denotes a type or subtype:

This attribute denotes the base type of P. It is only allowed as the prefix of the name of another attribute: for example, P'BASE'FIRST. (See 3.3.3.)

♦

P'FIRST — For a prefix P that denotes a scalar type, or a subtype of a scalar type:

Yields the lower bound of P. The value of this attribute has the same type as P. (See 3.5.)

P'FIRST — For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the lower bound of the first index range. The value of this attribute has the same type as this lower bound. (See 3.6.2 and 3.8.2.)

P'FIRST(N) — For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the lower bound of the N-th index range. The value of this attribute has the same type as this lower bound. The argument N must be a static expression of type *universal_integer*. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)

♦

P'IMAGE — For a prefix P that denotes a discrete type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the predefined type STRING. The result is the *image* of the value of X, that is, a sequence of characters representing the value in display form. The image of an integer value is the corresponding decimal literal; without underlines, leading zeros, exponent, or trailing spaces; but with a one character prefix that is either a minus sign or a space.

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. The image of a character other than a graphic character is implementation-defined. (See 3.5.5.)

♦

P'LAST — For a prefix P that denotes a scalar type, or a subtype of a scalar type:

Yields the upper bound of P. The value of this attribute has the same type as P. (See 3.5.)

P'LAST — For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the upper bound of the first index range. The value of this attribute has the same type as this upper bound. (See 3.6.2 and 3.8.2.)

P'LAST(N) — For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the upper bound of the N-th index range. The value of this attribute has the same type as this upper bound. The argument N must be a static expression of type *universal_integer*. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)

♦

P'LENGTH — For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the number of values of the first index range (zero for a null range). The value of this attribute is of the type *universal_integer*. (See 3.6.2.)

P'LENGTH(N) — For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the number of values of the N-th index range (zero for a null range). The value of this attribute is of the type *universal_integer*. The argument N must be a static expression of type *universal_integer*. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)

♦

P'POS — For a prefix P that denotes a discrete type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the type *universal_integer*. The result is the position number of the value of the actual parameter. (See 3.5.5.)

♦

P'PRED — For a prefix P that denotes a discrete type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one less than that of X. The exception CONSTRAINT_ERROR is raised if X equals P'BASE'FIRST. (See 3.5.5.)

P'RANGE — For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the first index range of P, that is, the range P'FIRST .. P'LAST. (See 3.6.2.)

P'RANGE(N) — For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the N-th index range of P, that is, the range P'FIRST(N) .. P'LAST(N). (See 3.6.2.)

♦

P'SUCC — For a prefix P that denotes a discrete type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one greater than that of X. The exception CONSTRAINT_ERROR is raised if X equals P'BASE'LAST. (See 3.5.5.)

♦

P'VAL — For a prefix P that denotes a discrete type or subtype:

This attribute is a special function with a single parameter X which can be of any integer type. The result type is the base type of P. The result is the value whose position number is the *universal_integer* value corresponding to X. The exception CONSTRAINT_ERROR is raised if the *universal_integer* value corresponding to X is not in the range P'POS(P'BASE'FIRST) .. P'POS(P'BASE'LAST). (See 3.5.5.)

P'VALUE For a prefix P that denotes a discrete type or subtype: 49

This attribute is a function with a single parameter. The actual parameter X must be a value of the predefined type STRING. The result type is the base type of P. Any leading and any trailing spaces of the sequence of characters that corresponds to X are ignored.

For an enumeration type, if the sequence of characters has the syntax of an enumeration literal and if this literal exists for the base type of P, the result is the corresponding enumeration value. For an integer type, if the sequence of characters has the syntax of an integer literal, with an optional single leading character that is a plus or minus sign, and if there is a corresponding value in the base type of P, the result is this value. In any other case, the exception CONSTRAINT_ERROR is raised. (See 3.5.5.)

P'WIDTH For a prefix P that denotes a discrete subtype: 50

Yields the maximum image length over all values of the subtype P (the *image* is the sequence of characters returned by the attribute IMAGE). The value of this attribute is of the type *universal_integer*. (See 3.5.5.)

Appendix B

Predefined Language Pragmas: Removed

Appendix C
Predefined Language Environment

This annex outlines the specification of the package STANDARD containing all predefined identifiers in the language. The corresponding package body is implementation-defined and is not shown. 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 and for undefined information (such as *implementation_defined* ♦). 2

```
package STANDARD is

  type BOOLEAN is (FALSE, TRUE);

  -- The predefined relational operators for this type are as follows:

  -- 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;
  -- function ">="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

  -- The predefined logical operators and the predefined logical negation operator are as follows:

  -- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "or"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

  -- function "not" (RIGHT : BOOLEAN) return BOOLEAN;

  -- The universal type universal_integer is predefined.

  type INTEGER is implementation_defined;

  -- The predefined operators for this type are as follows:

  -- function "="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function "<"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function ">"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function "+"   (RIGHT : INTEGER) return INTEGER;
  -- function "-"   (RIGHT : INTEGER) return INTEGER;
  -- function "abs" (RIGHT : INTEGER) return INTEGER;

  -- function "+"   (LEFT, RIGHT : INTEGER) return INTEGER;
  -- function "-"   (LEFT, RIGHT : INTEGER) return INTEGER;
  -- function "*"   (LEFT, RIGHT : INTEGER) return INTEGER;
  -- function "/"   (LEFT, RIGHT : INTEGER) return INTEGER;
  -- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
  -- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

  -- function "**"  (LEFT : INTEGER; RIGHT : INTEGER) return INTEGER;
```

```
-- An implementation may NOT provide additional predefined integer types. ♦                    7
-- The  specification of each operator for the type universal_integer♦
-- is obtained by replacing INTEGER by the
-- name of the type in the specification  of  the corresponding operator of the type
-- INTEGER, except for the right operand of the exponentiating operator.

♦

type FLOAT, REAL is limited private;                                                            9

-- The following characters form the standard ASCII character set.  Character literals        12
-- corresponding to control characters are not identifiers; they are indicated in italics in
-- this definition.

type CHARACTER is                                                                               13
   (    nul,    soh,    stx,    etx,    eot,    enq,    ack,    bel,
        bs,     ht,     lf,     vt,     ff,     cr,     so,     si,
        dle,    dc1,    dc2,    dc3,    dc4,    nak,    syn,    etb,
        can,    em,     sub,    esc,    fs,     gs,     rs,     us,

        ' ',    '!',    '"',    '#',    '$',    '%',    '&',    ''',
        '(',    ')',    '*',    '+',    ',',    '-',    '.',    '/',
        '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
        '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',

        '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
        'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
        'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
        'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',

        '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
        'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
        'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
        'x',    'y',    'z',    '{',    '|',    '}',    '~',    del);

   -- Enumeration representation clauses are not allowable AVA,
   -- but the following captures the intent of the data type.

   -- for CHARACTER use  --  128 ASCII character set without holes
   --      (0, 1, 2, 3, 4, 5, ..., 125, 126, 127);


   -- The predefined operators for the type CHARACTER are the same as for                      14
   --  any enumeration type.

package ASCII is                                                                                15

   -- Control characters:

   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:

    EXCLAM      : constant CHARACTER := '!';    QUOTATION   : constant CHARACTER := '"';
    SHARP       : constant CHARACTER := '#';    DOLLAR      : constant CHARACTER := '$';
    PERCENT     : constant CHARACTER := '%';    AMPERSAND   : constant CHARACTER := '&';
    COLON       : constant CHARACTER := ':';    SEMICOLON   : constant CHARACTER := ';';
    QUERY       : constant CHARACTER := '?';    AT_SIGN     : constant CHARACTER := '@';
    L_BRACKET   : constant CHARACTER := '[';    BACK_SLASH  : constant CHARACTER := '\';
    R_BRACKET   : constant CHARACTER := ']';    CIRCUMFLEX  : constant CHARACTER := '^';
    UNDERLINE   : constant CHARACTER := '_';    GRAVE       : constant CHARACTER := '`';
    L_BRACE     : constant CHARACTER := '{';    BAR         : constant CHARACTER := '|';
    R_BRACE     : constant CHARACTER := '}';    TILDE       : constant CHARACTER := '~';

  -- Lower case letters:

    LC_A : constant CHARACTER := 'a';
    ...
    LC_Z : constant CHARACTER := 'z';

end ASCII;
```

-- Predefined subtypes: 16

```
subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
```

-- Predefined string type: 17

```
type STRING is array(POSITIVE range <>) of CHARACTER;
```

♦

-- The predefined operators for this type are as follows: 18

```
-- 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, RIGHT : STRING) return BOOLEAN;
-- function "&" (LEFT : CHARACTER; RIGHT : STRING)    return STRING;
-- function "&" (LEFT : STRING;    RIGHT : CHARACTER) return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : CHARACTER) return STRING;

-- function "&" (LEFT : STRING;    RIGHT : STRING)    return STRING;
```

♦

-- The predefined exceptions: 20

♦

```
PROGRAM_ERROR    : exception;
```

♦

end STANDARD;

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. 21

Note:

The language definition predefines the following library units: 22

- The package AVA (see 13.7)
- The package AVA_IO (see 14.3.10)

Appendix D
Glossary

[This appendix is not part of the standard definition of the AVA *programming language.]*

This appendix is informative and is not part of the standard definition of the AVA programming language. Italicized terms in the abbreviated descriptions below either have glossary entries themselves or are described in entries for related terms.

♦

Actual parameter. See *parameter*.

Aggregate. The evaluation of an aggregate yields a value of a *composite type*. The value is specified by giving the value of each of the *components*. Either *positional association* or *named association* may be used to indicate which value is associated with which component.

ARM. *Reference Manual for the Ada Programming Language*.

Array type. A value of an array type consists of *components* which are all of the same *subtype* (and hence, of the same type). Each component is uniquely distinguished by an *index* (for a one-dimensional array) or by a sequence of indices (for a multidimensional array). Each index must be a value of a *discrete type* and must lie in the correct index *range*.

Assignment. Assignment is the *operation* that replaces the current value of a *variable* by a new value. An *assignment statement* specifies a variable on the left, and on the right, an *expression* whose value is to be the new value of the variable.

Attribute. The evaluation of an attribute yields a predefined characteristic of a named entity; some attributes are *functions*.

Block statement. A block statement is a single statement that may contain a sequence of statements. It may also include a *declarative part*, and *exception handlers*; their effects are local to the block statement.

Body. A body defines the execution of a *subprogram* or *package*.

♦

Compilation unit. A compilation unit is the *declaration* or the *body* of a *program unit*, presented for compilation as an independent text. It is optionally preceded by a *context clause*, naming other compilation units upon which it depends by means of one more *with clauses*.

Component. A component is a value that is a part of a larger value, or an *object* that is part of a larger object. A *component type* referes to the type of one of the subparts of a composite type.

Composite type. A composite type is one composed from component types, whose values have *components*. There are two kinds of composite type: *array types* and *record types*.

Constant. See *object*.

Constraint. A constraint determines a subset of the values of a *type*. A value in that subset *satisfies* the constraint.

Context clause. See *compilation unit*.

Declaration. A declaration associates an identifier (or some other notation) with an entity. This association is in effect within a region of text called the *scope* of the declaration. Within the scope of a declaration, there are places where it is possible to use the identifier to refer to the associated declared entity. At such places the identifier is said to be a *simple name* of the entity; the *name* is said to *denote* the associated entity.

Declarative Part. A declarative part is a sequence of *declarations*. It may also contain related information such as *subprogram bodies* and *representation clauses*.

Denote. See *declaration*.

♦

Direct visibility. See *visibility*.

Discrete Type. A discrete type is a *type* which has an ordered set of distinct values. The discrete types are the *enumeration* and *integer types*. Discrete types are used for indexing and iteration, and for choices in case statements ♦.

♦

Elaboration. The elaboration of a *declaration* is the process by which the declaration achieves its effect (such as creating an *object*); this process occurs during program execution.

♦

Enumeration type. An enumeration type is a *discrete type* whose values are represented by enumeration literals which are given explicitly in the *type declaration*. These enumeration literals are either *identifiers* or *character literals*.

Erroneous Execution. The Ada language definition specifies certain rules to be obeyed by Ada programs that do not need to be checked by Ada compilers either at compilation-time or run-time. The errors of this category are indicated by the use of the word *erroneous* to qualify the execution of the corresponding constructs. The effect of erroneous execution on the Ada program state is unpredictable. Erroneous programs have been eliminated from the AVA subset, either by eliminating the constructs that give rise to them or by specifying a required check. See also **incorrect order dependencies**.

Evaluation. The evaluation of an *expression* is the process by which the value of the expression is computed. This process occurs during program execution.

Exception. An exception is an error situation which may arise during program execution. To *raise* an exception is to abandon normal program execution so as to signal that the error has taken place. An

exception handler is a portion of program text specifying a response to the exception. Execution of such a program text is called *handling* the exception.

Expanded name. An expanded name *denotes* an entity which is *declared* immediately within some construct. An expanded name has the form of a *selected component*: the *prefix* denotes the construct (a *program unit*; or a *block*, or loop)♦; the *selector* is the *simple name* of the entity.

Expression. An expression defines the computation of a value.

♦

Formal parameter. See *parameter*.

Function. See *subprogram*.

♦

Handler. See *exception*.

Incorrect Order Dependencies, **IODs**. Whenever the Ada reference manual specifies that different parts of a given construct are to be executed *in some order that is not defined by the language*, this means that the implementation is allowed to execute these parts in any given order, following the rules that result from that given order, but not in parallel. The foregoing is expressed in terms of the process that is called execution; it applies equally to the processes that are called evaluation and elaboration. In Ada, the construct is incorrect if execution of these parts in a different order would have a different effect. In AVA we have ruled out IODs and erroneous programs by either eliminating the constructs that give rise to them or specifying a specific order of evalaution. These specific orders give rise to implementation requirements for compilers or front-ends that operate on AVA programs.

Index. See *array type*.

Index constraint. An index constraint for an *array type* specifies the lower and upper bounds for each index *range* of the array type.

Indexed component. An indexed component *denotes* a *component* in an *array*. It is a form of *name* containing *expressions* which specify the values of the *indices* of the array component. ♦

♦

Integer type. An integer type is a *discrete type* whose values represent all integer numbers within a specific *range*.

IODs. See **Incorrect Order Dependencies**.

Lexical element. A lexical element is an identifier, a *literal*, a delimiter, or a comment.

Limited type. A limited type is a *type* for which neither assignment nor the predefined comparison for equality is implicitly declared. ♦ A *private type* can be defined to be limited. An equality operator can be explicitly declared for a limited type.

Literal. A literal represents a value literally, that is, by means of letters and other characters. A literal is either a numeric literal, an enumeration literal, a character literal, or a string literal.

Mode. See *parameter*.

♦

Mutually accessible variable. A variable is *accessible* with respect to a particular expression if it is a subexpression of the expression or is accessed by calls on user functions within the expression. A *mutually* accessible variable is one that is accessible to two expressions that may be evaluated in an order not defined by the language.

Name. A name is a construct that stands for an entity: it is said that the name *denotes* the entity, and that the entity is the meaning of the name. See also *declaration*, *prefix*.

♦

Object. An object contains a value. A program creates an object ♦ by *elaborating* an *object declaration*. The declaration ♦ specifies a *type* for the object: the object can only contain values of that type.

Operation. An operation is an elementary action associated with one or more *types*. It is either implicitly declared by the *declaration* of the type, or it is a *subprogram* that has a *parameter* or *result* of the type.

Operator. An operator is an operation which has one or two operands. A unary operator is written before an operand; a binary operator is written between two operands. This notation is a special kind of *function call*. An operator can be declared as a function. Many operators are implicitly declared by the *declaration* of a *type* (for example, most type declarations imply the declaration of the equality operator for values of the type).

Overloading. An identifier can have several alternative meanings at a given point in the program text: this property is called *overloading*. For example, an overloaded enumeration literal can be an identifier that appears in the definitions of two or more *enumeration types*. The effective meaning of an overloaded identifier is determined by the context. *Subprograms*, *aggregates*, and string *literals* can also be overloaded.

Package. A package specifies a group of logically related entities, such as *types*, *objects* of those types, and *subprograms* with *parameters* of those types. It is written as a *package declaration* and a *package body*. The package declaration has a *visible part*, containing the *declarations* of all entities that can be explicitly used outside the package. It may also have a *private part* containing structural details that complete the specification of the visible entities, but which are irrelevant to the user of the package. The *package body* contains implementations of *subprograms* ♦ that have been specified in the package declaration. A package is one of the kinds of *program unit*.

Parameter. A parameter is one of the named entities associated with a *subprogram* ♦ and used to communicate with the corresponding subprogram body ♦. A *formal parameter* is an *identifier* used to denote the named entity within the body. An *actual parameter* is the particular entity associated with the corresponding formal parameter by a *subprogram call* ♦. The *mode* of a formal parameter specifies whether the associated actual parameter supplies a value for the formal parameter, or the formal supplies a value for the actual parameter, or both. The association of actual parameters with formal parameters can be specified by ♦ *positional associations* ♦.

♦

Positional association. A positional association specifies the association of an item with a position in a list, by using the same position in the text to specify the item.

♦

Prefix. A prefix is used as the first part of certain kinds of name. A prefix is either a *function call* or a *name*.

Private part. See *package*.

Private type. A private type is a *type* whose structure and set of values are clearly defined, but not directly available to the user of the type. A private type is known only by ♦ the set of *operations* defined for it. A private type and its applicable operations are defined in the *visible part* of a *package* ♦. *Assignment*, equality, and inequality are also defined for private types, unless the private type is *limited*.

Procedure. See *subprogram*.

Program. A program is composed of a number of *compilation units*, one of which is a *subprogram* called the *main program*. Execution of the program consists of execution of the main program, which may invoke subprograms declared in the other compilation units of the program.

Program unit. A program unit is any one of ♦ a *package* or *subprogram*.

Qualified expression. A qualified expression is an *expression* preceded by an indication of its *type* or *subtype*. Such qualification is used when, in its absence, the expression might be ambiguous (for example as a consequence of *overloading*).

Raising an exception. See *exception*.

Range. A range is a contiguous set of values of a *scalar type*. A range is specified by giving the lower and upper bounds for the values. A value in the range is said to *belong* to the range.

Range constraint. A range constraint of a *type* specifies a *range*, and thereby determines the subset of the values of the type that *belong* to the range.

♦

Record type. A value of a record type consists of *components* which are usually of different *types* or *subtypes*. For each component of a record value or record *object*, the definition of the record type specifies an identifier that uniquely determines the component within the record.

Renaming declaration. A renaming declaration declares another *name* for an entity.

♦

Satisfy. See *constraint, subtype*.

Scalar type. An *object* or value of a scalar *type* does not have *components*. A scalar type is ♦ a *discrete*

type♦. The values of a scalar type are ordered.

Scope. See *declaration*.

Selected component. A selected component is a *name* consisting of a *prefix* and of an identifier called the *selector*. Selected components are used to denote record components ♦; they are also used as *expanded names*.

Selector. See *selected component*.

Simple name. See *declaration, name*.

Statement. A statement specifies one or more actions to be performed during the execution of a *program*.

Subcomponent. A subcomponent is either a *component*, or a component of another subcomponent.

Subprogram. A subprogram is either a *procedure* or a *function*. A procedure specifies a sequence of actions and is invoked by a *procedure call* statement. A function specifies a sequence of actions and also returns a value called the *result*, and so a *function call* is an *expression*. A subprogram is written as a *subprogram declaration*, which specifies its *name*, *formal parameters*, and (for a function) its result; and a *subprogram body* which specifies the sequence of actions. The subprogram call specifies the *actual parameters* that are to be associated with the formal parameters. A subprogram is one of the kinds of *program unit*.

Subtype. A subtype of a *type* characterizes a subset of the values of the type. The subset is determined by a *constraint* on the type. Each value in the set of values of a subtype *belongs* to the subtype and *satisfies* the constraint determining the subtype.

♦

Type. A type characterizes both a set of values, and a set of *operations* applicable to those values. A *type definition* is a language construct that defines a type. A particular type is either an ♦ an *array type*, a *private type*, a *record type*, or a *scalar type* ♦.

Use clause. A use clause achieves *direct visibility* of *declarations* that appear in the *visible parts* of named *packages*.

Variable. See *object*.

♦

Visibility. At a given point in a program text, the *declaration* of an entity with a certain identifier is said to be *visible* if the entity is an acceptable meaning for an occurrence at that point of the identifier. The declaration is *visible* by *selection* at the place of the *selector* in a *selected component* or at the place of the name in a *named association*. Otherwise, the declaration is *directly visible*, that is, if the identifier alone has that meaning.

Visible part. See *package*.

With clause. See *compilation unit*.

Appendix E
Syntax Summary

[This appendix is not part of the standard definition of the AVA *programming language.]*

2.1

```
graphic_character ::= basic_graphic_character
  | lower_case_letter | other_special_character

basic_graphic_character ::=
    upper_case_letter | digit
  | special_character | space_character

basic_character ::=
    basic_graphic_character | format_effector
```

2.3

```
identifier ::=
  letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter
```

2.4

```
numeric_literal ::= decimal_literal | based_literal
```

2.4.1

```
decimal_literal ::= integer [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer
```

2.4.2

```
based_literal ::=
  base # based_integer # [exponent]

base ::= integer

based_integer ::=
  extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter
```

2.5

```
character_literal ::= 'graphic_character'
```

2.6

```
string_literal ::= "{graphic_character}"
```

3.1

```
basic_declaration ::=
    inner_declaration
  | type_declaration      | subtype_declaration
  | subprogram_declaration | package_declaration  | ♦
  | renaming_declaration   | deferred_constant_declaration

inner_declaration ::=
    object_declaration    | number_declaration
```

3.2

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

number_declaration ::=
    identifier_list : constant := universal_static_expression;

identifier_list ::=  identifier {, identifier}
```

3.3.1

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

full_type_declaration ::=
 type identifier is type_definition;

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

3.3.2

```
subtype_declaration ::=
  subtype identifier is subtype_indication;

subtype_indication ::=  type_mark [constraint]

type_mark ::= type_name | subtype_name

constraint ::=
    range_constraint
  | index_constraint
  | ♦
```

3.4

♦

3.5

```
range_constraint ::=  range range
```

```
range ::= range_attribute
  | simple_expression .. simple_expression

3.5.1

enumeration_type_definition ::=
  (enumeration_literal_specification
    {, enumeration_literal_specification})

enumeration_literal_specification ::= enumeration_literal

enumeration_literal ::= identifier | character_literal

3.5.4

♦

3.6

array_type_definition ::=
    unconstrained_array_definition
  | constrained_array_definition

unconstrained_array_definition ::=
  array(index_subtype_definition
        {, index_subtype_definition}) of
        component_subtype_indication

constrained_array_definition ::=
  array index_constraint of
        component_subtype_indication

  index_subtype_definition ::= index_type_mark range <>

  index_constraint ::= (index_range {, index_range})

  index_range ::= integer_subtype_indication | integer_range

  discrete_range ::= discrete_subtype_indication | range

3.7

record_type_definition ::=
  record
    component_list
  end record

component_list ::=
    component_declaration {component_declaration}
  | null;

component_declaration ::=
  identifier_list : component_subtype_definition;

component_subtype_definition ::= subtype_indication

3.8.1

♦

3.9

declarative_part ::=
  {basic_declarative_item} {later_declarative_item}

basic_declarative_item ::= basic_declaration
  | ♦ | use_clause

later_declarative_item ::= subprogram_body | package_body
    | subprogram_declaration | package_declaration |♦
    | use_clause          | ♦

inner_declarative_part ::= {inner_declarative_item}

4.1

name ::= simple_name
  | character_literal  | operator_symbol
  | indexed_component  |
  | selected_component | attribute

simple_name ::= identifier

prefix ::= name | function_call

4.1.1

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

4.1.3

selected_component ::= prefix.selector

selector ::= simple_name ♦

4.1.4

attribute ::= prefix'attribute_designator
attribute_designator ::=
  simple_name [(universal_static_expression)]

4.3

aggregate ::=
  (component_association {, component_association})

component_association ::=
  [ choice {| choice} = ] expression

choice ::= simple_expression | discrete_range
      | component_simple_name | others

4.4

expression ::=
    relation {and relation} | relation {and then relation}
  | relation {or relation}  | relation {or else relation}
  | relation {xor relation}

relation ::=
    simple_expression [relational_operator simple_expression]
  | simple_expression [not] in range
  | simple_expression [not] in type_mark

simple_expression ::=
```

```
  [unary_adding_operator] term
  {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=
    numeric_literal | ♦ | aggregate | string_literal
  | name | function_call | type_conversion
  | qualified_expression | (expression)
```

4.5

```
logical_operator  ::=  and | or | xor

relational_operator  ::=  = | /= | < | <= | > | >=

binary_adding_operator  ::=  + | - | &

unary_adding_operator  ::=  + | -

multiplying_operator  ::=  * | / | mod | rem

highest_precedence_operator  ::=  ** | abs | not
```

4.6

```
type_conversion ::= type_mark(expression)
```

4.7

```
qualified_expression ::=
  type_mark'(expression) | type_mark'aggregate
```

5.1

```
sequence_of_statements ::= statement {statement}

statement ::=
    ♦ simple_statement
  | ♦ compound_statement

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

compound_statement ::=
    if_statement     | case_statement
  | loop_statement   | block_statement

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;

condition ::= boolean_expression
```

5.4

```
case_statement ::=
  case expression is
      case_statement_alternative
    {case_statement_alternative}
  end case;

case_statement_alternative ::=
  when choice {| choice } =>
    sequence_of_statements
```

5.5

```
loop_statement ::=
  [loop_simple_name:]
    [iteration_scheme] loop
      sequence_of_statements
    end loop [loop_simple_name];

iteration_scheme ::= while condition
  | for loop_parameter_specification

loop_parameter_specification ::=
  identifier in [reverse] discrete_range
```

5.6

```
block_statement ::= ♦
    [declare
       inner_declarative_part]
     begin
       sequence_of_statements
    [exception
       exception_handler ♦]
     end ♦;
```

5.7

```
exit_statement ::=
  exit ♦ [when condition];
```

5.8

```
return_statement ::= return [expression];
```

6.1

```
subprogram_declaration ::= subprogram_specification;

subprogram_specification ::=
    procedure identifier [formal_part]
  | function identifier [formal_part] return type_mark
```

♦

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

parameter_specification ::=
  identifier_list : mode type_mark ♦

mode ::= [in] | in out | ♦
```

6.3

```
subprogram_body ::=
   subprogram_specification is
     [inner_declarative_part]
   begin
      sequence_of_statements
  [exception
      exception_handler ♦]
   end [identifier];
```

6.4

```
procedure_call_statement ::=
  procedure_name [actual_parameter_part];

function_call ::=
  function_name [actual_parameter_part]

actual_parameter_part ::=
  (parameter_association {, parameter_association})

parameter_association ::= actual_parameter

♦
actual_parameter ::=
    expression | variable_name
  | ♦
```

7.1

```
package_declaration ::= package_specification;

package_specification ::=
   package identifier is
     {basic_declarative_item}
  [private
     {basic_declarative_item}]
   end [package_simple_name]

package_body ::=
   package body package_simple_name is
     [declarative_part]
  [begin
      sequence_of_statements
  [exception
      exception_handler ♦}]]
   end [package_simple_name];
```

7.4

```
private_type_declaration ::=
  type identifier is ♦ private;

deferred_constant_declaration ::=
  identifier_list : constant type_mark;
```

8.4

```
use_clause ::= use package_name {, package_name};
```

8.5

```
renaming_declaration ::=
    identifier : type_mark   renames object_name;
  | ♦
  | package identifier       renames package_name;
  | subprogram_specification renames subprogram_name;
```

10.1

```
compilation ::= {compilation_unit}

compilation_unit ::=
    context_clause library_unit
  | context_clause secondary_unit

library_unit ::=
    subprogram_declaration | package_declaration
  | subprogram_body

secondary_unit ::= library_unit_body

library_unit_body ::= subprogram_body | package_body
```

10.1.1

```
context_clause ::= {with_clause {use_clause}}

with_clause ::=
  with unit_simple_name {, unit_simple_name};
```

10.2 : Removed

11.1

♦

11.2

```
exception_handler ::=
  when others =>
    sequence_of_statements

exception_choice ::= exception_name | others
```

11.3

```
raise_statement ::= raise program_error;
```

Syntax Cross Reference

abs ...
  factor 4.4
  highest_precedence_operator 4.5

actual_parameter 6.4
  parameter_association 6.4

actual_parameter_part 6.4
  function_call 6.4
  procedure_call_statement 6.4

aggregate 4.3
  primary 4.4
  qualified_expression 4.7

all ...
  selector 4.1.3

and ...
  expression 4.4
  logical_operator 4.5

argument_association 2.8

array ...
  constrained_array_definition 3.6
  unconstrained_array_definition 3.6

array_type_definition 3.6
  type_definition 3.3.1

assignment_statement 5.2
  simple_statement 5.1

attribute 4.1.4
  name 4.1
  range 3.5

attribute_designator 4.1.4
  attribute 4.1.4

base 2.4.2
  based_literal 2.4.2

based_integer 2.4.2
  based_literal 2.4.2

based_literal 2.4.2
  numeric_literal 2.4

basic_character 2.1

basic_declaration 3.1
  basic_declarative_item 3.9

basic_declarative_item 3.9
  declarative_part 3.9
  package_specification 7.1

basic_graphic_character 2.1
  basic_character 2.1
  graphic_character 2.1

begin ...
  block_statement 5.6
  package_body 7.1
  subprogram_body 6.3

binary_adding_operator 4.5
  simple_expression 4.4

block_statement 5.6
  compound_statement 5.1

body 3.9
  later_declarative_item 3.9

body ...
  package_body 7.1

case ...
  case_statement 5.4

case_statement 5.4
  compound_statement 5.1

case_statement_alternative 5.4
  case_statement 5.4

character_literal 2.5
  enumeration_literal 3.5.1
  name 4.1
  selector 4.1.3

choice 4.3
  case_statement_alternative 5.4
  component_association 4.3

compilation 10.1

compilation_unit 10.1
  compilation 10.1

component_association 4.3
  aggregate 4.3

component_declaration 3.7
  component_list 3.7

component_list 3.7
  record_type_definition 3.7

component_subtype_definition 3.7
  component_declaration 3.7

compound_statement 5.1
  statement 5.1

condition 5.3
  exit_statement 5.7
  if_statement 5.3
  iteration_scheme 5.5

constant ...
deferred_constant_declaration 7.4
number_declaration 3.2
object_declaration 3.2

constrained_array_definition 3.6
array_type_definition 3.6
object_declaration 3.2

constraint 3.3.2
subtype_indication 3.3.2

context_clause 10.1.1
compilation_unit 10.1

decimal_literal 2.4.1
numeric_literal 2.4

declarative_part 3.9
block_statement 5.6
package_body 7.1
subprogram_body 6.3

declare ...
block_statement 5.6

deferred_constant_declaration 7.4
basic_declaration 3.1

designator 6.1
subprogram_body 6.3
subprogram_specification 6.1

digit ...
basic_graphic_character 2.1
extended_digit 2.4.2
integer 2.4.1
letter_or_digit 2.3

discrete_range 3.6
choice 4.3
index_constraint 3.6
loop_parameter_specification 5.5

E ...
exponent 2.4.1

else ...
expression 4.4
if_statement 5.3

elsif ...
if_statement 5.3

end ...
block_statement 5.6
case_statement 5.4
if_statement 5.3
loop_statement 5.5
package_body 7.1
package_specification 7.1
record_type_definition 3.7
subprogram_body 6.3

enumeration_literal 3.5.1
enumeration_literal_specification 3.5.1

enumeration_literal_specification 3.5.1
enumeration_type_definition 3.5.1

enumeration_type_definition 3.5.1
type_definition 3.3.1

exception ...
block_statement 5.6
exception_declaration 11.1
package_body 7.1
subprogram_body 6.3

exception_choice 11.2
exception_handler 11.2

exception_declaration 11.1
basic_declaration 3.1

exception_handler 11.2
block_statement 5.6
package_body 7.1
subprogram_body 6.3

exit ...
exit_statement 5.7

exit_statement 5.7
simple_statement 5.1

exponent 2.4.1
based_literal 2.4.2
decimal_literal 2.4.1

expression 4.4
actual_parameter 6.4
argument_association 2.8
assignment_statement 5.2
attribute_designator 4.1.4
case_statement 5.4
component_association 4.3
component_declaration 3.7
condition 5.3
indexed_component 4.1.1
number_declaration 3.2
object_declaration 3.2
parameter_specification 6.1
primary 4.4
qualified_expression 4.7
return_statement 5.8
type_conversion 4.6

extended_digit 2.4.2
based_integer 2.4.2

factor 4.4
term 4.4

for ...
iteration_scheme 5.5

formal_parameter 6.4

parameter_association 6.4

formal_part 6.1
subprogram_specification 6.1

format_effector ...
basic_character 2.1

full_type_declaration 3.3.1
type_declaration 3.3.1

function ...
subprogram_specification 6.1

function_call 6.4
prefix 4.1
primary 4.4

graphic_character 2.1
character_literal 2.5
string_literal 2.6

highest_precedence_operator 4.5

identifier 2.3
argument_association 2.8
enumeration_literal 3.5.1
full_type_declaration 3.3.1
identifier_list 3.2
loop_parameter_specification 5.5
package_specification 7.1
private_type_declaration 7.4
renaming_declaration 8.5
simple_name 4.1
subprogram_specification 6.1
subtype_declaration 3.3.2

identifier_list 3.2
component_declaration 3.7
deferred_constant_declaration 7.4
exception_declaration 11.1
number_declaration 3.2
object_declaration 3.2
parameter_specification 6.1

if ...
if_statement 5.3

if_statement 5.3
compound_statement 5.1

in ...
loop_parameter_specification 5.5
mode 6.1
relation 4.4

index_constraint 3.6
constrained_array_definition 3.6
constraint 3.3.2

index_subtype_definition 3.6
unconstrained_array_definition 3.6

indexed_component 4.1.1
name 4.1

integer 2.4.1
base 2.4.2
decimal_literal 2.4.1
exponent 2.4.1

integer_type_definition 3.5.4
type_definition 3.3.1

is ...
case_statement 5.4
full_type_declaration 3.3.1
package_body 7.1
package_specification 7.1
private_type_declaration 7.4
subprogram_body 6.3
subtype_declaration 3.3.2

iteration_scheme 5.5
loop_statement 5.5

later_declarative_item 3.9
declarative_part 3.9

letter 2.3
extended_digit 2.4.2
identifier 2.3
letter_or_digit 2.3

letter_or_digit 2.3
identifier 2.3

library_unit 10.1
compilation_unit 10.1

library_unit_body 10.1
secondary_unit 10.1

♦

logical_operator 4.5

loop ...
loop_statement 5.5

loop_parameter_specification 5.5
iteration_scheme 5.5

loop_statement 5.5
compound_statement 5.1

lower_case_letter ...
graphic_character 2.1
letter 2.3

mod ...
multiplying_operator 4.5

mode 6.1
parameter_specification 6.1

multiplying_operator 4.5
term 4.4

name 4.1
actual_parameter 6.4
argument_association 2.8
assignment_statement 5.2
component_clause 13.4
exception_choice 11.2
exit_statement 5.7
function_call 6.4
prefix 4.1
primary 4.4
procedure_call_statement 6.4
raise_statement 11.3
renaming_declaration 8.5
type_mark 3.3.2
use_clause 8.4

not ...
factor 4.4
highest_precedence_operator 4.5
relation 4.4

null ...
♦
null_statement 5.1
primary 4.4

null_statement 5.1
simple_statement 5.1

number_declaration 3.2
basic_declaration 3.1

numeric_literal 2.4
primary 4.4

object_declaration 3.2
basic_declaration 3.1

of ...
constrained_array_definition 3.6
unconstrained_array_definition 3.6

operator_symbol 6.1
designator 6.1
name 4.1
selector 4.1.3

or ...
expression 4.4
logical_operator 4.5

other_special_character ...
graphic_character 2.1

others ...
choice 4.3
exception_choice 11.2

out ...
mode 6.1

package ...
package_body 7.1
package_specification 7.1
renaming_declaration 8.5

package_body 7.1
library_unit_body 10.1
proper_body 3.9

package_declaration 7.1
basic_declaration 3.1
later_declarative_item 3.9
library_unit 10.1

package_specification 7.1
package_declaration 7.1

parameter_association 6.4
actual_parameter_part 6.4

parameter_specification 6.1
formal_part 6.1

prefix 4.1
attribute 4.1.4
indexed_component 4.1.1
selected_component 4.1.3

primary 4.4
factor 4.4

private ...
package_specification 7.1
private_type_declaration 7.4

private_type_declaration 7.4
type_declaration 3.3.1

procedure ...
subprogram_specification 6.1

procedure_call_statement 6.4
simple_statement 5.1

proper_body 3.9
body 3.9

qualified_expression 4.7
allocator 4.8
primary 4.4

raise ...
raise_statement 11.3

raise_statement 11.3
simple_statement 5.1

range 3.5
discrete_range 3.6
range_constraint 3.5
relation 4.4

range ...
index_subtype_definition 3.6
range_constraint 3.5

range_constraint 3.5

constraint 3.3.2
♦

record ...
record_type_definition 3.7

record_type_definition 3.7
type_definition 3.3.1

relation 4.4
expression 4.4

relational_operator 4.5
relation 4.4

rem ...
multiplying_operator 4.5

renames ...
renaming_declaration 8.5

renaming_declaration 8.5
basic_declaration 3.1

return ...
return_statement 5.8
subprogram_specification 6.1

return_statement 5.8
simple_statement 5.1

reverse ...
loop_parameter_specification 5.5

secondary_unit 10.1
compilation_unit 10.1

selector 4.1.3
selected_component 4.1.3

sequence_of_statements 5.1
block_statement 5.6
case_statement_alternative 5.4
exception_handler 11.2
if_statement 5.3
loop_statement 5.5
package_body 7.1
subprogram_body 6.3

simple_expression 4.4
choice 4.3
range 3.5
relation 4.4

simple_name 4.1
attribute_designator 4.1.4
block_statement 5.6
choice 4.3
formal_parameter 6.4
loop_statement 5.5
name 4.1
package_body 7.1
package_specification 7.1
selector 4.1.3
with_clause 10.1.1

simple_statement 5.1
statement 5.1

space_character ...
basic_graphic_character 2.1

special_character ...
basic_graphic_character 2.1

statement 5.1
sequence_of_statements 5.1

string_literal 2.6
operator_symbol 6.1
primary 4.4

subprogram_body 6.3
library_unit 10.1
library_unit_body 10.1
proper_body 3.9

subprogram_declaration 6.1
basic_declaration 3.1
later_declarative_item 3.9
library_unit 10.1

subprogram_specification 6.1
renaming_declaration 8.5
subprogram_body 6.3
subprogram_declaration 6.1

subtype ...
subtype_declaration 3.3.2

subtype_declaration 3.3.2
basic_declaration 3.1

subtype_indication 3.3.2
component_subtype_definition 3.7
constrained_array_definition 3.6
discrete_range 3.6
object_declaration 3.2
subtype_declaration 3.3.2
unconstrained_array_definition 3.6

term 4.4
simple_expression 4.4

then ...
expression 4.4
if_statement 5.3

type ...
full_type_declaration 3.3.1
private_type_declaration 7.4

type_conversion 4.6
primary 4.4

type_declaration 3.3.1
basic_declaration 3.1

type_definition 3.3.1
full_type_declaration 3.3.1

type_mark 3.3.2
actual_parameter 6.4
deferred_constant_declaration 7.4
index_subtype_definition 3.6
parameter_specification 6.1
qualified_expression 4.7
relation 4.4
renaming_declaration 8.5
subprogram_specification 6.1
subtype_indication 3.3.2
type_conversion 4.6

unary_adding_operator 4.5
simple_expression 4.4

unconstrained_array_definition 3.6
array_type_definition 3.6

underline ...
based_integer 2.4.2
identifier 2.3
integer 2.4.1

upper_case_letter ...
basic_graphic_character 2.1
letter 2.3

use ...
use_clause 8.4

use_clause 8.4
basic_declarative_item 3.9
context_clause 10.1.1
later_declarative_item 3.9

when ...
case_statement_alternative 5.4
exception_handler 11.2
exit_statement 5.7

while ...
iteration_scheme 5.5

with ...
with_clause 10.1.1

with_clause 10.1.1
context_clause 10.1.1

xor ...
expression 4.4
logical_operator 4.5

" ...
string_literal 2.6

...
based_literal 2.4.2

& ...
binary_adding_operator 4.5

' ...
attribute 4.1.4
character_literal 2.5

◆
qualified_expression 4.7

( ) ...
actual_parameter 6.4
actual_parameter_part 6.4
aggregate 4.3
attribute_designator 4.1.4
enumeration_type_definition 3.5.1
formal_part 6.1
index_constraint 3.6
indexed_component 4.1.1
primary 4.4
qualified_expression 4.7
type_conversion 4.6
unconstrained_array_definition 3.6

* ...
multiplying_operator 4.5

** ...
factor 4.4
highest_precedence_operator 4.5

+ ...
binary_adding_operator 4.5
exponent 2.4.1
unary_adding_operator 4.5

, ...
actual_parameter_part 6.4
aggregate 4.3
discriminant_constraint 3.7.2
enumeration_type_definition 3.5.1
identifier_list 3.2
index_constraint 3.6
indexed_component 4.1.1
unconstrained_array_definition 3.6
use_clause 8.4
with_clause 10.1.1

- ...
binary_adding_operator 4.5
exponent 2.4.1
unary_adding_operator 4.5

. ...
based_literal 2.4.2
decimal_literal 2.4.1
selected_component 4.1.3

.. ...
range 3.5

/ ...
multiplying_operator 4.5

/= ...
relational_operator 4.5

: ...

block_statement 5.6
component_declaration 3.7
deferred_constant_declaration 7.4
exception_declaration 11.1
loop_statement 5.5
number_declaration 3.2
object_declaration 3.2
parameter_specification 6.1
renaming_declaration 8.5

:= ...
assignment_statement 5.2
component_declaration 3.7
number_declaration 3.2
object_declaration 3.2

; ...
assignment_statement 5.2
block_statement 5.6
case_statement 5.4
component_declaration 3.7
component_list 3.7
deferred_constant_declaration 7.4
exception_declaration 11.1
exit_statement 5.7
formal_part 6.1
full_type_declaration 3.3.1
if_statement 5.3
loop_statement 5.5
null_statement 5.1
number_declaration 3.2
object_declaration 3.2
package_body 7.1
package_declaration 7.1
private_type_declaration 7.4
procedure_call_statement 6.4
raise_statement 11.3
renaming_declaration 8.5
return_statement 5.8
subprogram_body 6.3
subprogram_declaration 6.1
subtype_declaration 3.3.2
use_clause 8.4
with_clause 10.1.1

< ...
relational_operator 4.5

<= ...
relational_operator 4.5

<> ...
index_subtype_definition 3.6

= ...
relational_operator 4.5

=> ...
case_statement_alternative 5.4
component_association 4.3
exception_handler 11.2
parameter_association 6.4

> ...
relational_operator 4.5

>= ...
relational_operator 4.5

| ...
case_statement_alternative 5.4
component_association 4.3
exception_handler 11.2

Appendix F
Implementation-Dependent Characteristics

This appendix is not part of the standard definition of the AVA *programming language.*

The Ada language definition allows for certain machine-dependences in a controlled manner. No machine-dependent syntax or semantic extensions or restrictions are allowed. The only allowed implementation-dependences correspond to implementation-dependent pragmas and attributes, certain machine-dependent conventions as mentioned in chapter 13, and certain allowed restrictions on representation clauses. Since most of these are ruled out of AVA, the list of implementation specific features is short. 23

The reference manual of each AVA implementation must include an appendix (called Appendix F) that describes all implementation-dependent characteristics. The appendix F for a given implementation must list in particular: 24

(1) Omitted 25

(2) The name and the type of every implementation-dependent attribute. 26

(3) The specification of the package AVA_SYSTEM (see 13). 27

(4) Omitted 28

(5) The conventions used for any implementation-generated name denoting implementation-dependent components (see 13.4). 29

(6) Omitted 30

(7) Omitted 31

(8) Any implementation-dependent characteristics of the input-output packages (see 14). 32

Appendix G
INDEX

This appendix is not part of the standard definition of the Ada programming language.

Abandoning a declaration or sequence of statements
[see: exception, raise statement]

Abnormal termination of a subprogram call 6.2

Abs unary operator 4.5, 4.5.6
[see also: highest precedence operator]
as an operation 3.3.3
as an operation of an integer type 3.5.5
in a factor 4.4
propagating an exception 11.6
raising an exception 11.4.1

Absolute value operation 4.5.6

Actual parameter 6.4.1; D; (of an operator) 6.7; (of a subprogram) 6.4; 6.2, 6.3
[see also: formal parameter, function call, procedure call statement, subprogram call]
characteristics and overload resolution 6.6
of an array type 3.6.1
of a record type 3.7.2
which is an array aggregate 4.3.2
which is a loop parameter 5.5

Actual parameter part 6.4
in a function call 6.4
in a procedure call statement 6.4

Adding operator
[see: binary adding operator, unary adding operator]

Addition operation 4.5.3

Aggregate 4.3, D
[see also: array aggregate, overloading of..., record aggregate]
as a basic operation 3.3.3, 4.3; 3.6.2, 3.7.4
as a primary 4.4
in a qualified expression 4.7
may not be the argument of a conversion 4.6

Aliasing 6.2

Allowed 1.6

Alternative
[see: case statement alternative]

Ambiguities between overloaded subprograms 6.6

Ambiguous
[see: overloading]

Ampersand
[see: catenation]
character 2.1
delimiter 2.2

Ancestor library unit 10.2

And operator
[see: logical operator]

And then control form
[see: short circuit control form]

Anonymous type 3.3.1
anonymous base type [see: first named subtype]
anonymous base type due to derivation 3.4

ANSI (american national standards institute) 2.1

Apostrophe character 2.1
in a character literal 2.5

Apostrophe delimiter 2.2
[see also: identifier after...]
in an attribute 4.1.4
of a qualified expression 4.7

Appropriate for a type 4.1
for an array type 4.1.1, 4.1.2
for a record type 4.1.3
prefix of an attribute 4.1.4

Arithmetic operator 4.5; 3.5.5
[see also: binary adding operator, exponentiating operator, multiplying operator, predefined operator, unary adding operator]
as an operation of an integer type 3.5.5

Array aggregate 4.3.2; 4.3, D
[see also: aggregate]

Array assignment 5.2.1

Array bounds
[see: bounds of an array]

Array component
[see: array type, component, indexed component]

Array type 3.6; 3.3, D
[see also: component, composite type, constrained array, constrained..., index, matching components, unconstrained...]
appropriate for a prefix of an indexed component 4.1.1
as a full type 7.4.1
conversion 4.6

formal parameter which is of an array type 6.2
indexing 4.1.1
operation 3.6.2; 4.5.2, 4.5.3
operation on an array of boolean components 4.5.1, 4.5.6
subject to an attribute 3.6.2
with a limited component type 7.4.4

Array type definition 3.6; 3.3.1, 12.1.2, 12.3.4
[see also: constrained array definition, elaboration of..., unconstrained array definition]

Arrow compound delimiter 2.2

ASCII (american standard code for information interchange) 2.1

ASCII (predefined library package) 3.5.2; 2.6, C
[see also: graphical symbol]

Assignment compound delimiter 2.2; 5.2
in an object declaration 3.2.1

Assignment operation 5.2; D
[see also: limited type]
as a basic operation 3.3, 3.3.3; 3.5.5, 3.5.8, 3.6.2, 3.7.4, 3.8.2, 7.4.2, 12.1.2
not available for a limited type 7.4.4
of an array aggregate 4.3.2
of an initial value to an object 3.2.1
to an array variable 5.2.1; 5.2
to a loop parameter 5.5

Assignment statement 5.2; D
[see also: statement]
as a simple statement 5.1

Association
[see: component association, parameter association]

Association of a declarative region with a declaration or statement 8.1

Attribute 4.1.4; D
[see also: predefined attribute, predefined function]
as a basic operation 3.3.3; 3.5.8, 3.6.2
as a basic operation of a discrete type 3.5.5
as a name 4.1
as a primary 4.4
of a discrete type or subtype 3.5.5
of a static subtype in a static expression 4.9
of a type 3.3
renamed as a function 8.5
starting with a prefix 4.1, 4.1.4

Attribute designator 4.1.4
in an attribute 4.1.4

Bar
[see: vertical bar]

BASE (predefined attribute) 3.3.3; A
for an array type 3.6.2
for a discrete type 3.5.5
for a private type 7.4.2
for a record type 3.7.4

Base type (of a subtype) 3.3
as a static subtype 4.9
as target type of a conversion 4.6
due to elaboration of a type definition 3.3.1
name [see: name of a base type]
of an array type 3.6; 4.1.2
of an integer type 3.5.4
of an integer type is an anonymous predefined type 3.5.4
of a qualified expression 4.7
of a type mark 3.3.2
of a type mark in a membership test 4.5.2
of the discrete range in a loop parameter specification 5.5
of the expression in a case statement 5.4
of the result subtype of a function 5.8
of the type mark in a renaming declaration 8.5

Based literal 2.4.2
[see also: colon character, sharp character]
as a numeric literal 2.4

Basic character 2.1
[see also: basic graphic character, character]

Basic character set 2.1
is sufficient for a program text 2.10

Basic declaration 3.1
as a basic declarative item 3.9

Basic declarative item 3.9
in a package specification 7.1; 7.2

Basic graphic character 2.1
[see also: basic character, digit, graphic character, space character, special character, upper case letter]

Basic loop 5.5

Basic operation 3.3.3
[see also: aggregate, assignment, conversion direct visibility, operation, scope of..., string literal, visibility by selection, visibility]
implicitly declared 3.1, 3.3.3
of an array type 3.6.2
of a discrete type 3.5.5
of a limited type 7.4.4
of a private type 7.4.2
of a record type 3.7.4
propagating an exception 11.6
raising an exception 11.4.1
which is an attribute 4.1.4

Becomes compound delimiter
[see: assignment compound delimiter]

Belong

to a range 3.5
to a subtype 3.3

Binary adding operator 4.5; 4.5.3, C
[see also: adding operation, arithmetic operator, overloading of an operator]
in a simple expression 4.4
overloaded 6.7

Binary operation 4.5

Blank skipped by a text_io procedure 14.3.5

Block as an entity 3.1

Block name 5.6
declaration 5.1
implicitly declared 3.1

Block statement 5.6; D
[see also: completed block statement, statement]
as a compound statement 5.1
as a declarative region 8.1
entity denoted by an expanded name 4.1.3
including an exception handler 11.2; 11
including an implicit declaration 5.1
raising an exception 11.4.1, 11.4.2

Body 3.9; D
[see also: declaration, library unit, package body, proper body, subprogram body]
as a later declarative item 3.9

BOOLEAN (predefined type) 3.5.3; C
result of an attribute 9.9
result of a condition 5.3
result of an explicitly declared equality operator 6.7

Boolean expression
[see: relation]

Boolean operator
[see: relational operator]

Boolean types 3.5.3
[see also: predefined type]
operation 3.5.5; 4.5.1, 4.5.2, 4.5.6
subject to an attribute 3.5.5

Bound
[see: error bound, first attribute, last attribute]

Bound of an array 3.6, 3.6.1
[see also: index range]
aggregate 4.3.2
result of a logical operator 4.5.1
result of an operation 4.5.1, 4.5.3, 4.5.6
which is a formal parameter 6.2

Bound of a range 3.5; 3.5.4
of a discrete range is of universal_integer type 3.6.1
of a static discrete range 4.9

Bound of a range constraint

Bound of a subtype
of a static discrete subtype 4.9
returned by first and last attributes 3.5.5, 3.5.10

Box compound delimiter 2.2
in an index subtype definition 3.6

Bracket
[see: label bracket, left parenthesis, parenthesised expression, right parenthesis, string bracket]

Call
[see: function call, procedure call statement, subprogram call]

Carriage return format effector 2.1

Case of a letter
[see: letter, lower case letter, upper case letter]

Case statement 5.4
[see also: statement]
as a compound statement 5.1

Case statement alternative 5.4

Catenation operation 4.5.3
for a character type 3.5.5
in a replacement of a string literal 2.10

Catenation operator 4.5; 2.6, 3.6.3, 4.5.3, C
[see also: predefined operator]
for an array type 3.6.2

Character 2.1; D
[see also: ampersand, apostrophe, basic character, colon, divide, dot, equal, exclamation mark character, graphic character, greater than, hyphen, less than, minus, other special character, parenthesis, percent, period, plus, point character, pound sterling, quotation, semicolon, sharp, space, special character, star, underline, vertical bar]
in a lexical element 2, 2.2
names of characters 2.1
replacement in program text 2.10

CHARACTER (predefined type) 3.5.2; C
as the component type of the string type 3.6.3

Character literal 2.5, 4.2; 2.2, 3.5.2
[see also: direct visibility, predefined function, scope of..., space character literal, visibility by selection, visibility]
as a basic operation 3.3.3
as an enumeration literal 3.5.1
as a name 4.1
declared by an enumeration literal specification 3.1
in a selector 4.1.3

in a static expression 4.9
in homograph declarations 8.3
must be visible at the place of a string literal 4.2

Character types 3.5.2; 2.5
operation 3.5.5
subject to an attribute 3.5.5

Choice 3.7.3
[see also: exception choice]
in an aggregate 4.3
in an array aggregate 4.3.2
in a case statement alternative 5.4
in a component association 4.3, 4.3.1, 4.3.2
in a record aggregate 4.3.1

Circularity in dependencies between compilation units 10.5

Class of type 3.3; 12.1.2
[see also: composite type, private type, scalar type]

Clause
[see: use clause, with clause]

CLOSE (input-output procedure)
in text_io 14.2.1; 14.3.10

Closed file 14.1

COL (text_io function) 14.3.4; 14.3.10
raising an exception 14.4

Colon character 2.1
[see also: based literal]
replacing sharp character 2.10

Colon delimiter 2.2

Column 14.3.4

Comma character 2.1

Comma delimiter 2.2

Comment 2.7; 2.2
in a code procedure body 13.8
in a conforming construct 6.3.1

Comparison
[see: relational operator]

Compatibility (of constraints) 3.3.2
[see also: constraint]
failure not causing constraint_error 11.7
of a discrete range with an index subtype 3.6.1
of index constraints 3.6.1
of range constraints 3.5

Compilation 10.1; 10, 10.4
as a sequence of lexical elements 2

Compilation order
[see: order of compilation]

Compilation unit 10.1; D, 10, 10.4
[see also: library unit, secondary unit]
compiled after library units named in its context clause 10.3
with a context clause 10.1.1
with a use clause 8.4

Compilation-time evaluation of expressions 4.9

Compile time evaluation of expressions 10.6

Compiler 10.4

Completed block statement 9.4

Completed subprogram 9.4

Component (of a composite type) 3.3; 3.6, 3.7, D
[see also: component association, component list, composite type, indexed component, object, record type, selected component, subcomponent]
as an entity 3.1
combined by aggregate 4.3
name starting with a prefix 4.1
of an array 3.6 [see also: array type, component, indexed component]
of a constant 3.2.1
of an object 3.2
of a private type 7.4.2
of a record 3.7 [see also: component, record type, selected component]
of a variable 3.2.1
simple name as a choice 3.7.3
subtype indication 3.7
subtype itself an array type 3.6.1
which is of limited type 7.4.4

Component association 4.3
in an aggregate 4.3
including an expression which is an array aggregate 4.3.2
positional component association 4.3

Component declaration 3.7
[see also: declaration, record type definition]
as part of a basic declaration 3.1
having an extended scope 8.2
in a component list 3.7
of an array object 3.6.1
of a record object 3.7.2
visibility 8.3

Component list 3.7
in a record type definition 3.7

Component subtype definition 3.7
in a component declaration 3.7

Component type
catenation with an array type 4.5.3

object initialization [see: initial value]
of an expression in an array aggregate 4.3.2
of an expression in a record aggregate 4.3.1
operation determining a composite type
operation 4.5.1, 4.5.2

Composite type 3.3; 3.6, 3.7
[see also: array type, class of type, component, record type, subcomponent]
including a limited subcomponent 7.4.4
object initialization 3.2.1 [see also: initial value]
of an aggregate 4.3
with a private type component 7.4.2

Compound delimiter 2.2
[see also: arrow, assignment, box, delimiter, double dot, double star, exponentiation, greater than or equal, inequality, left label bracket, less than or equal, right label bracket]
names of delimiters 2.2

Compound statement 5.1
[see also: statement]

Concatenation
[see: catenation]

Condition 5.3
[see also: expression]
as a boolean expression 3.5.3
in an exit statement 5.7
in an if statement 5.3
in a while iteration scheme 5.5

Conditional compilation 10.6

Conflicting names resolved 8.5

Conforming
formal parts 6.3.1
subprogram specifications 6.3.1; 6.3
type marks 6.3.1; 7.4.3

Conjunction
[see: logical operator]

Constant 3.2.1; D
[see also: deferred constant, loop parameter, object]
formal parameter 6.2
in a static expression 4.9
renamed 8.5

Constant declaration 3.2.1
[see also: deferred constant declaration]
as a full declaration 7.4.3
with an array type 3.6.1
with a record type 3.7.2

Constrained array definition 3.6
in an object declaration 3.2, 3.2.1

Constrained array type 3.6
[see also: array type, constraint]

Constrained subtype 3.3; 3.2.1, 3.6, 3.6.1, 3.7, 3.7.2, 6.4.1
[see also: constraint, subtype, type, unconstrained subtype]
due to elaboration of a type definition 3.3.1
object declarations 3.2.1

Constrained variable 3.7

Constraint (on an object of a type) 3.3, 3.3.2; D
[see also: compatibility, constrained subtype, elaboration of..., index constraint, range constraint, satisfy, subtype, unconstrained subtype]
explicitly specified by use of a qualification 4.7
not considered in overload resolution 8.7
on a formal parameter 6.2
on a renamed object 8.5
on a subcomponent subject to a component clause must be static 13.4
on a variable 3.2.1, 3.3, 3.6

CONSTRAINT_ERROR
(predefined exception) 11.1
[see also: index_check, range_check]
raised by an actual parameter not in the subtype of the formal parameter 6.4.1
raised by an assignment 5.2; 3.5.4
raised by an attribute 3.5.5
raised by a component of an array aggregate 4.3.2
raised by a component of a record aggregate 4.3.1
raised by a formal parameter not in the subtype of the actual parameter 6.4.1
raised by an index value out of bounds 4.1.1, 4.1.2
raised by a logical operation on arrays of different lengths 4.5.1
raised by a name with a prefix evaluated to null 4.1
raised by a qualification 4.7
raised by a result of a conversion 4.6
raised by a return statement 5.8
raised by incompatible constraints 3.3.2
raised by integer exponentiation with a negative exponent 4.5.6
raised by matching failure in an array assignment 5.2.1
raised by the initialization of an object 3.2.1
raised by the result of catenation 4.5.3
suppressed 11.7

Context clause 10.1.1; D
[see also: use clause, with clause]
determining order of elaboration of compilation units 10.5
in a compilation unit 10.1
including a use clause 8.4
inserted by the environment 10.4

Context of overload resolution 8.7
[see also: overloading]

Control form
[see: short circuit control form]

Conversion
[see: numeric type, subtype conversion, type conversion, unchecked conversion]
in a static expression 4.9
of a universal type expression 5.2 [see also: universal type expression]
of the bounds of a loop parameter 5.5

Conversion operation 4.6
[see also: explicit conversion, implicit conversion, subtype conversion, type conversion]
as a basic operation 3.3.3; 3.3, 3.5.5, 3.5.8, 3.6.2, 3.7.4, 3.8.2, 7.4.2
between numeric types 3.3.3, 3.5.5

Convertible universal operand 4.6

Copy parameter passing 6.2

COUNT (predefined integer type) 14.2, 14.2.5, 14.3.10; 14.2.4, 14.3, 14.3.3, 14.3.4, 14.4

Current column number 14.3; 14.3.1, 14.3.4, 14.3.5, 14.3.6

Current line number 14.3; 14.3.1, 14.3.4, 14.3.5

Current mode of a file 14.1, 14.2.1; 14.2.2, 14.2.4, 14.3, 14.3.5, 14.4

Current page number 14.3; 14.3.1, 14.3.4, 14.3.5

CURRENT_INPUT
(text_io function) 14.3.2; 14.3.10

CURRENT_OUTPUT
(text_io function) 14.3.2; 14.3.10

DATA_ERROR (input-output exception) 14.4; 14.2.2, 14.2.3, 14.2.4, 14.2.5, 14.3.5, 14.3.7, 14.3.8, 14.3.9, 14.3.10, 14.5

Dead code elimination 10.6

Decimal literal 2.4.1
as a numeric literal 2.4

Decimal point
[see: point character]

Declaration 3.1; D
[see also: basic declaration, block name declaration, body, component declaration, constant declaration, deferred constant declaration, denote, enumeration literal specification, exception declaration, exception raised during..., hiding, implicit declaration, incomplete type declaration, label declaration, loop name declaration, loop parameter specification, number declaration, object declaration, package declaration, package specification, parameter specification, private type declaration, renaming declaration, scope of..., specification, subprogram declaration, subprogram specification, subtype declaration, type declaration, visibility]
as an overload resolution context 8.7
determined by visibility from an identifier 8.3
made directly visible by a use clause 8.4
of an enumeration literal 3.5.1
of a formal parameter 6.1
of a loop parameter 5.5
optimized 10.6
overloaded 6.6
raising an exception 11.4.2; 11.4

Declarative item 3.9
[see also: basic declarative item, later declarative item]
in a code procedure body 13.8
in a declarative part 3.9; 6.3.2
in a package specification 6.3.2
in a visible part 7.4
which is a use clause 8.4

Declarative part 3.9; D
[see also: elaboration of..., elaboration]
in a block statement 5.6
in a package body 7.1; 7.3
in a subprogram body 6.3
including a specification and its body 9.1
with implicit declarations 5.1

Declarative region 8.1; 8.2, 8.4, D
[see also: scope of...]
determining the visibility of a declaration 8.3
formed by the predefined package standard 8.6
in which a declaration is hidden 8.3
including a full type definition 7.4.2
including a subprogram declaration 6.3

Declared immediately within
[see: occur immediately within]

Default file 14.3.2; 14.3

Default mode (of a file) 14.2.1; 14.2.3, 14.2.5, 14.3.10

Deferred constant 7.4.3
of a limited type 7.4.4

Deferred constant declaration 7.4; 7.4.3
[see also: private part (of a package), visible part (of a package)]
as a basic declaration 3.1

Definition
[see: array type definition, component subtype definition, constrained array definition, enumeration type definition, index subtype definition, integer type definition, record type definition, type definition, unconstrained array definition]

Delimiter 2.2
[see also: ampersand, apostrophe delimiter, arrow, assignmenty, colon, compound delimiter, divide, dor, dot delimiter, double dot, equal, exclamation mark delimiter, exponentiation, greater than, greather than or equal, inequality, label bracket, less than or equal, less than, minus, parenthesis, period, plus, point, semicolon delimiter, semicolon, star, vertical bar delimiter, vertical bar]

Denote an entity 3.1, 4.1; D
[see also: declaration, entity, name]

Dependence between compilation units 10.3; 10.5
[see also: with clause]
circularity implying illegality 10.5

Designate 3.8; D
[see also: object designated by...]

Designator
[see: attribute designator, name of a function, name of an operator, name of a subprogram, operator, overloading of ...]; (of a function) 6.1
in a function declaration 4.5
in a subprogram body 6.3
in a subprogram specification 6.1; 6.3
of a library unit 10.1
overloaded 6.6

DEVICE_ERROR (input-output exception) 14.4; 14.2.3, 14.2.5, 14.3.10, 14.5

Digit 2.1
[see also: basic graphic character, extended digit, letter or digit]
in a based literal 2.4.2
in a decimal literal 2.4.1
in an identifier 2.3

Dimensionality of an array 3.6

Direct input-output 14.2.4; 14.2.1

Direct visibility 8.3
[see also: basic operation, character literal, operation, operator symbol, selected component, visibility]
due to a use clause 8.4
of a library unit due to a with clause 10.1.1

Discrete range 3.6, 3.6.1
[see also: range, static discrete range, static...]
as a choice 3.7.3
as a choice in an aggregate 4.3
for a loop parameter 5.5
in a choice in a case statement 5.4
in an index constraint 3.6
in a loop parameter specification 5.5

Discrete type 3.5; D
[see also: basic operation of..., enumeration type, index, integer type, iteration scheme, operation of..., scalar type]
expression in a case statement 5.4
of a loop parameter 5.5
of index values of an array 3.6
operation 3.5.5; 4.5.2
subject to an attribute 3.5.5

Disjunction
[see: logical operator]

Divide
character 2.1
delimiter 2.2

Division operation 4.5.5

Division operator
[see: multiplying operator]

Division_check
[see: numeric_error, suppress]

Dot
[see: double dot]
character 2.1 [see also: double dot, point character]
delimiter 2.2
delimiter of a selected component 8.3; 4.1.3

Double dot compound delimiter 2.2

Double hyphen starting a comment 2.7

Double star compound delimiter 2.2
[see also: exponentiation compound delimiter]

Effect
[see: no other effect]

ELABORATE (predefined pragma) 10.5; B

Elaborated 3.9

Elaboration 3.9; 3.1, 3.3, D, 10.1
[see also: exception raised during..., no other effect, order of elaboration]
optimized 10.6

Elaboration of
an array type definition [see: elaboration of a constrained array definition]
a component declaration 3.7
a component subtype specification 3.7
a constrained array definition 3.6; 3.2.1
a context clause 10.1
a declaration 3.1
a declaration raising an exception 11.4
a declarative item 3.9
a declarative part 3.9; 5.6, 6.3
a declarative part raising an exception 11.4.2
a deferred constant declaration 7.4.3
an enumeration type definition 3.5.1
a formal part 9.5
a full type declaration 3.8.1, 7.4.1

an incomplete type declaration 3.8.1
an index constraint 3.6.1; 3.6
an integer type definition 3.5.4
a loop parameter specification 5.5
a number declaration 3.2.2
an object declaration 3.2.1; 9.2
a package body 7.3
a package body raising an exception 11.4.2
a package body terminated by an exception 11.4.1, 11.4.2
a package declaration 7.2
a package declaration raising an exception 11.4.2
a private type declaration 7.4.1
a range constraint 3.5.4
a record type definition 3.7
a renaming declaration 8.5
a subprogram body 6.3; 7.3
a subprogram declaration 6.1
a subtype declaration 3.3.2
a subtype indication 3.3.2; 3.2.1, 3.4, 3.6, 3.7, 3.8, 4.8, 11.7
a subtype of an object 3.2.1
a type declaration 3.3.1
a type definition 3.3.1; 3.3, 7.4.1
an unconstrained array definition 3.6
a use clause 8.4

Elaboration_check
[see: program_error exception, suppress]

Element in a file 14, 14.1; 14.2

Else part
of an if statement 5.3

Empty string literal 2.6

End of line 2.2
as a separator 2.2
due to a format effector 2.2
terminating a comment 2.7

END_ERROR (input-output exception) 14.4; 14.2.2, 14.2.3, 14.2.4, 14.2.5, 14.3.4, 14.3.5, 14.3.6, 14.3.10, 14.5

END_OF_FILE (input-output function)
in text_io 14.3.1, 14.3.10

END_OF_LINE (text_io function) 14.3.4; 14.3.10
raising an exception 14.4

END_OF_PAGE (text_io function) 14.3.4; 14.3.10, 14.4

Entity 3.1; 3.2, D
denoted by a name 4.1

Enumeration literal 3.5.1, 4.2; D
[see also: overloading of..., predefined function]
as an operation 3.3.3
as an operator 3.5.5
as result for image attribute 3.5.5
as the parameter for value attribute 3.5.5
implicitly declared 3.3.3
in a static expression 4.9
overloaded 8.3
renamed as a function 8.5

Enumeration literal specification 3.5.1
as part of a basic declaration 3.1
made directly visible by a use clause 8.4

Enumeration type 3.5.1; 3.3, 3.5, D
[see also: discrete type, scalar type]
as a character type 3.5.2
boolean 3.5.3
operation 3.5.5
subject to an attribute 3.5.5

Enumeration type definition 3.5.1; 3.3.1
[see also: elaboration of...]

Environment of a program 10.4
environment task calling the main program 10.1

Equal
character 2.1
delimiter 2.2

Equality operation 4.5.2
for a limited type 4.5.2
not available for a limited type 7.4.4

Equality operator 4.5; 4.5.2
[see also: limited type, relational operator]
explicitly declared 4.5.2, 6.7; 7.4.4
for an array type 3.6.2
for a record type 3.7.4

Erroneous 1.6
[see also: program_error]

Erroneousness due to
an unchecked conversion violating properties of objects of the result type 13.10.2
dependence between initial values of subcomponents 3.2.1
dependence on parameter-passing mechanism 6.2
suppression of an exception check 11.7

Error detected at
compilation time 1.6
run time 1.6

Error situation 1.6, 11, 11.1; 11.6

Error that may not be detected 1.6

Evaluation (of an expression) 4.5
[see also: compile time evaluation, expression]
at compile time 10.6
of an actual parameter 6.4.1
of an aggregate 4.3; 3.3.3
of an array aggregate 4.3.2
of a condition 3.5.3, 5.3
of a discrete range 3.5; 9.5
of a discrete range used in an index

constraint 3.6.1
of an expression in an assignment statement 5.2
of an expression in a constraint 3.3.2
of an indexed component 4.1.1
of an initial value
of a literal 4.2; 3.3.3
of a logical operation 4.5.1
of a name 4.1; 4.1.1, 4.1.2, 4.1.3, 4.1.4
of a name in a renaming declaration 8.5
of a primary 4.4
of a qualified expression 4.7; 4.8
of a range 3.5
of a record aggregate 4.3.1
of a short circuit control form 4.5.1
of a static expression 4.9
of a type conversion 4.6
of a universal expression 4.10
of the bounds of a loop parameter 5.5
of the name of a variable in an assignment statement 5.2

Evaluation order
[see: order of evaluation]

Exception 11; 1.6, D
[see also: constraint_error, name of..., numeric_error, predefined .., program_error, raise statement, raising of ..]
as an entity 3.1
causing a loop to be exited 5.5
causing a transfer of control 5.1
due to an expression evaluated at compile time 10.6
in input-output 14.4; 14.5
renamed 8.5

Exception choice 11.2

Exception declaration 11.1; 11
as a basic declaration 3.1

Exception handler 11.2; D
in a block statement 5.6
in a package body 7.1; 7.3
in a subprogram body 6.3
including a raise statement 11.3
including the name of an exception 11.1
not allowed in a code procedure body 13.8
raising an exception 11.4.1
selected to handle an exception 11.4.1; 11.6

Exception handling 11.4; 11.4.1, 11.4.2, 11.5

Exception propagation 11
from a declaration 11.4.2
from a predefined operation 11.6
from a statement 11.4.1

Exception raised during
a declaration 11.4.2; 11.4
a raise statement 11.3
a statement 11.4.1; 11.4
a subprogram call 6.3; 6.2

Exclamation character 2.1
replacing vertical bar 2.10

Exclusive disjunction
[see: logical operator]

Execution
[see: sequence of statements, statement]

Exit statement 5.7
[see also: statement]
as a simple statement 5.1
causing a loop to be exited 5.5
causing a transfer of control 5.1
completing block statement execution 9.4

Expanded name 4.1.3; D
denoting a loop 5.5
in a static expression 4.9

Explicit conversion 4.6
[see also: conversion operation, implicit conversion, subtype conversion, type conversion]

Explicit declaration 3.1; 4.1
[see also: declaration]

Explicit initialization
[see: object declaration, qualified expression]

Exponent part
of a based literal 2.4.1, 2.4.2
of a decimal literal 2.4.1

Exponentiating operator 4.5; 4.5.6
[see also: highest precedence operator]
in a factor 4.4
overloaded 6.7

Exponentiation compound delimiter 2.2
[see also: double star compound delimiter]

Exponentiation operation 4.5.6

Expression 4.4; D
[see also: compile time evaluation, evaluation, qualified expression, simple expression, static expression, universal type expression]
as an actual parameter 6.4, 6.4.1
as a condition 5.3
in an assignment statement 5.2
in an attribute designator 4.1.4
in a case statement 5.4
in a case statement as an overload resolution context 8.7
in a choice in a case statement 5.4
in a component association 4.3
in a component declaration 3.7
in a constraint 3.3.2
in a conversion 4.6
in an indexed component 4.1.1
in a name 4.1
in a name of a variable 5.2, 6.4.1
in a number declaration 3.2

in an object declaration 3.2, 3.2.1
in a parameter specification 6.1
in a primary 4.4
in a qualified expression 4.7
in a return statement 5.8
in a type conversion as an overload resolution context 8.7
including the name of a private type 7.4.1
specifying the value of an index 4.1.1
with a boolean result [see: condition]

Extended_digit in a based literal 2.4.2

External file 14.1
[see also: file]

Factor 4.4
in a term 4.4

FALSE boolean enumeration literal 3.5.3; C

Feed
[see: form feed, line feed]

File (object of a file type) 14.1
[see also: external file]

File management 14.2.1
in text_io 14.3.1

File terminator 14.3; 14.3.1, 14.3.4, 14.3.5, 14.3.6, 14.3.7, 14.3.8, 14.3.9

FILE_MODE (input-output type)
in text_io 14.1, 14.2.1; 14.3.10

FILE_TYPE (input-output type)
in text_io 14.1, 14.2.1; 14.2, 14.3.3, 14.3.4, 14.3.6, 14.3.7, 14.3.8, 14.3.9, 14.3.10

FIRST (predefined attribute) A
[see also: bound]
for an array type 3.6.2
for a discrete type 3.5.5

First named subtype 13.1
[see also: anonymous base type]

Font design of graphical symbols 2.1

For loop
[see: loop statement]

Form feed format effector 2.1

Formal parameter 6.1; D; (of a function) 6.5; (of an operator) 6.7; (of a subprogram) 6.1, 6.2, 6.4; 3.2, 3.2.1, 6.3
[see: actual parameter, mode, object, subprogram]
as a constant 3.2.1
as an entity 3.1
as an object 3.2
as an object renamed 8.5
as a variable 3.2.1
characteristics and overload resolution 6.6
names and overload resolution 6.6
of a main program 10.1
of an operation 3.3.3
of a renamed subprogram 8.5
which is of an array type 3.6.1
which is of a limited type 7.4.4
which is of a record type 3.7.2

Formal part 6.1; 6.4
[see also: parameter type profile]
conforming to another 6.3.1
in a subprogram specification 6.1

Format effector 2.1
[see also: carriage return, form feed, horizontal tabulation, line feed, vertical tabulation]
as a separator 2.2
in an end of line 2.2

Format of text_io input or output 14.3.5, 14.3.7, 14.3.8, 14.3.9

Formula
[see: condition, expression]

Frame 11.2
in which an exception is raised 11.4.1, 11.4.2

Full declaration
of a deferred constant 7.4.3

Full type declaration 3.3.1
of an incomplete type 3.8.1
of a limited private type 7.4.4
of a private type 7.4.1; 7.4.2

Function 6.1, 6.5; 6, D, 12.3
[see also: name of a function, operator, parameter and result type profile, parameter, predefined function, result subtype, return statement, subprogram]
as a main program 10.1
renamed 8.5
result [see: returned value]
which is an attribute 4.1.4; 12.3.6

Function body
[see: subprogram body]

Function call 6.4; 6
[see also: actual parameter, subprogram call]
as a prefix 4.1
as a primary 4.4
in a static expression 4.9

Function specification
[see: subprogram specification]

Functional nature of an enumeration literal 3.5.1

GET (text_io procedure) 14.3.5; 14.3, 14.3.2,

14.3.4, 14.3.10
for character and string types 14.3.6
for enumeration types 14.3.9
for integer types 14.3.7
raising an exception 14.4

GET_LINE (text_io procedure) 14.3.6; 14.3.10

Global declaration 8.1

Graphic character 2.1
[see also: basic graphic character, character, lower case letter, other special character]
in a character literal 2.5
in a string literal 2.6

Graphical symbol 2.1
[see also: ascii]
not available 2.10

Greater than
character 2.1
delimiter 2.2
operator [see: relational operator]

Greater than or equal
compound delimiter 2.2
operator [see: relational operator]

Group of logically related entities
[see: package]

Handler
[see: exception handler, exception handling]

Hiding (of a declaration) 8.3
[see also: visibility]
and renaming 8.5
due to an implicit declaration 5.1
impossible due to a use clause 8.4
of a library unit 10.1
of a subprogram 6.6
of the package standard 10.1

Highest precedence operator 4.5
[see also: abs, arithmetic operator, exponentiating operator, not, overloading of an operator, predefined operator]
as an operation of a discrete type 3.5.5
overloaded 6.7

Homograph declaration 8.3
[see also: overloading]
preventing direct visibility due to a use clause 8.4

Horizontal tabulation
as a separator 2.2
character in a comment 2.7
format effector 2.1

Hyphen character 2.1
[see also: minus character]
starting a comment 2.7

Identifier 2.3; 2.2, D
[see also: direct visibility, loop parameter, name, overloading of..., scope of..., simple name, visibility]
and an adjacent separator 2.2
as an attribute designator 2.9, 4.1.4
as a designator 6.1
as a reserved word 2.9
as a simple name 4.1
can be written in the basic character set 2.10
denoting an object 3.2.1
denoting a value 3.2.2
in an attribute 4.1.4
in a deferred constant declaration 7.4.3
in an exception declaration 11.1
in an incomplete type declaration 3.8.1
in a number declaration 3.2.2
in an object declaration 3.2.1
in a package specification 7.1
in a private type declaration 7.4; 7.4.1
in a renaming declaration 8.5
in a subprogram specification 6.1
in a type declaration 3.3.1; 7.4.1
in its own declaration 8.3
of an enumeration value 3.5.1
of a library unit 10.1
of a subprogram 6.1
of a subtype 3.3.2
of homograph declarations 8.3
overloaded 6.6
versus simple name 3.1

Identifier list 3.2
in a component declaration, 3.7
in a deferred constant declaration 7.4
in a number declaration 3.2
in an object declaration 3.2
in a parameter specification 6.1

Identity operation 4.5.4

If statement 5.3
[see also: statement]
as a compound statement 5.1

IMAGE (predefined attribute) 3.5.5; A

Immediate scope 8.2; 8.3

Immediately within (a declarative region) 8.1; 8.3, 8.4, 10.2

Implementation dependent
[see: system dependent]

Implicit conversion 4.6
[see also: conversion operation, explicit conversion, subtype conversion]
of an integer literal to an integer type 3.5.4
of a universal type expression 3.5.4, 3.5.6

Implicit declaration 3.1; 4.1
[see also: scope of...]
at instantiation 12.3

by a type declaration 4.5
hidden by an explicit declaration 8.3
of a basic operation 3.1, 3.3.3
of a block name, loop name, or label 5.1; 3.1
of an enumeration literal 3.3.3
of an equality operator 6.7
of an exception due to an instantiation 11.1
of a library unit 8.6
of a predefined operator 4.5

In membership test
[see: membership test operation, membership test relation]

In mode
[see: mode in]

In out mode
[see: mode in out]

In some order that is not defined 1.6
[see also: incorrect order dependency, program_error]

IN_FILE (input-output file_mode enumeration literal) 14.1

Inclusive disjunction
[see: logical operator]

Incompatibility (of constraints)
[see: compatibility]

Incomplete type 3.8.1
corresponding full type declaration 3.3.1

Incomplete type declaration 3.8.1; 3.3.1, 7.4.1
as a portion of a declarative region 8.1

Incorrect order dependence 1.6; 3.5
[see also: program error]
between expressions of a record aggregate 4.3.1
between expressions or choices of an array aggregate 4.3.2
copying back of out and in out formal parameters 6.4
in expression evaluation 4.5, 5.2

Index 3.6; D
[see also: array, discrete type]

Index constraint 3.6, 3.6.1; D
ignored due to index_check suppression 11.7
in a constrained array definition 3.6
in a subtype indication 3.3.2
violated 11.1

Index range 3.6
matching 4.5.2

Index subtype 3.6

Index subtype definition 3.6

Index subtype definition in an unconstrained array definition 3.6

Index type
of a choice in an array aggregate 4.3.2

Index_check
[see: constraint_error, suppress]

Indexed component 4.1.1; 3.6, D
as a basic operation 3.3.3; 3.3, 3.6.2, 3.8.2
as a name 4.1

Indication
[see: subtype indication]

Inequality compound delimiter 2.2

Inequality operation 4.5.2
not available for a limited type 7.4.4

Inequality operator 4.5; 4.5.2
[see also: limited type, relational operator]
cannot be explicitly declared 6.7
for an array type 3.6.2
for a record type 3.7.4

Initial value (of an object) 3.2.1
[see also: composite type]
of an array object 3.6.1
of a constant 3.2.1
of a constant in a static expression 4.9
of a limited private type object 7.4.4
of an object declared in a package 7.1
of an out mode formal parameter 6.2
of a record object 3.7.2

Initialization
[see: assignment, initial value]

INOUT_FILE (input-output file_mode enumeration literal) 14.1

Input-output 14
[see also: io_exceptions, text_io]
at device level 14.6
exceptions 14.4; 14.5
with a text file 14.3

INTEGER (predefined type) 3.5.4; C
as base type of a loop parameter 5.5
as default type for the bounds of a discrete range 3.6.1; 9.5

Integer literal 2.4
[see also: based integer literal, universal_integer type]
as a bound of a discrete range 9.5
as a universal integer literal 3.5.4
in based notation 2.4.2
in decimal notation 2.4.1

Integer part
as a base of a based literal 2.4.2
of a decimal literal 2.4.1

Integer predefined types 3.5.4
[see also: INTEGER]

Integer subtype
[see: priority]
due to an integer type definition 3.5.4

Integer type declaration
[see: integer type definition]

Integer type definition 3.5.4; 3.3.1
[see also: elaboration of...]

Integer types 3.5.4; 3.3, 3.5, D
[see also: discrete type, numeric type, predefined type, scalar type, system.max_int, system.min_int, universal_integer type]
operation 3.5.5; 4.5.3, 4.5.4, 4.5.5, 4.5.6
result of a conversion from a numeric type 4.6
result of an operation out of range of the type 4.5
subject to an attribute 3.5.5

Interrupt 13.5

IO_EXCEPTIONS (predefined input-output package) 14.4; C, 14, 14.1, 14.2.3, 14.2.5, 14.3.10
specification 14.5

IS_OPEN (input-output function)
in text_io 14.2.1; 14.3.10

ISO (international organization for standards) 2.1

ISO seven bit coded character set 2.1

Item
[see: basic declarative item, later declarative item]

Iteration scheme 5.5
[see also: discrete type]

Labeled statement 5.1
as an entity 3.1

Language interface to other language systems 13.9

LARGE (predefined attribute) 3.5.8, 3.5.10; A

LAST (predefined attribute) A
[see also: bound]
for an array type 3.6.2
for a discrete type 3.5.5

Later declarative item 3.9

Layout recommended for programs 1.5

LAYOUT_ERROR (input-output exception) 14.4; 14.3.4, 14.3.5, 14.3.7, 14.3.8, 14.3.9, 14.3.10, 14.5

Leading zeros in a numeric literal 2.4.1

Left label bracket compound delimiter 2.2

Left parenthesis
character 2.1
delimiter 2.2

Legal 1.6

LENGTH (predefined attribute) 3.6.1, 3.6.2; A

Length of a string literal 2.6

Length of the result
of an array comparison 4.5.1
of an array logical negation 4.5.6
of a catenation 4.5.3

Length_check
[see: constraint_error, suppress]

Less than
character 2.1
delimiter 2.2
operator [see: relational operator]

Less than or equal
compound delimiter 2.2
operator [see: relational operator]

Letter 2.3
[see also: lower case letter, upper case letter]
e or E in a decimal literal 2.4.1
in a based literal 2.4.2
in an identifier 2.3

Letter_or_digit 2.3

Lexical element 2, 2.2; 2.4, 2.5, 2.6, D
as a point in the program text 8.3
in a conforming construct 6.3.1
transferred by a text_io procedure 14.3, 14.3.5, 14.3.9

Lexicographic order 4.5.2

Library package
[see: library unit, package]

Library package body
[see: library unit, package body]
raising an exception 11.4.1, 11.4.2

Library unit 10.1; 10.5
[see also: compilation unit, predefined package, predefined subprogram, program unit, standard predefined package]
compiled before the corresponding body 10.3
included in the predefined package standard 8.6
named in a use clause 10.5
named in a with clause 10.1.1; 10.3, 10.5
recompiled 10.3
scope 8.2
visibility due to a with clause 8.3
which is a package 7.1
whose names are needed in a compilation unit 10.1.1

Limited private type 7.4.4
[see also: private type]

Limited type 7.4.4; 9.2, D, 12.3.1
[see also: assignment, equality operator, inequality operator, predefined operator]
as a full type 7.4.1
component of a record 3.7
in an object declaration 3.2.1
limited record type 3.7.4
operation 7.4.4; 4.5.2
parameters for explicitly declared equality operators 6.7

Line 14.3.4; 14.3

LINE (text_io function) 14.3.4; 14.3.10
raising an exception 14.4

Line feed format effector 2.1

Line length 14.3.3; 14.3, 14.3.1, 14.3.4, 14.3.5, 14.3.6

Line terminator 14.3; 14.3.4, 14.3.5, 14.3.6, 14.3.7, 14.3.8, 14.3.9

LINE_LENGTH (text_io function) 14.3.3, 14.3.4; 14.3.3, 14.3.10
raising an exception 14.4

List
[see: component list, identifier list]

Literal 4.2; D
[see also: based literal, character literal, decimal literal, enumeration literal, integer literal, null literal, numeric literal, overloading of..., string literal]
as a basic operation 3.3.3
specification [see: enumeration literal specification]

Local declaration 8.1

Logical negation operation 4.5.6

Logical operation 4.5.1

Logical operator 4.5; 4.4, 4.5.1, C
[see also: overloading of an operator, predefined operator]
as an operation of boolean type 3.5.5
for an array type 3.6.2
in an expression 4.4
overloaded 6.7

Logical processor 9

Logically related entities
[see: package]

Loop
[see: basic loop]
as an entity 3.1

Loop name 5.5
declaration 5.1
implicitly declared 3.1
in an exit statement 5.7

Loop parameter 5.5
[see also: constant, object]
as an entity 3.1
as an object 3.2

Loop parameter specification 5.5
[see also: elaboration of...]
as an overload resolution context 8.7
is a declaration 3.1

Loop statement 5.5
[see also: statement]
as a compound statement 5.1
as a declarative region 8.1
entity denoted by an expanded name 4.1.3
including an exit statement 5.7

Lower bound
[see: bound, first attribute]

Lower case letter 2.1
[see also: graphic character]
a to f in a based literal 2.4.2
e in a decimal literal 2.4.1
in an identifier 2.3

Machine dependent attribute 13.7.3

Main program 10.1
execution requiring elaboration of library units 10.5
included in the predefined package standard 8.6
raising an exception 11.4.1, 11.4.2
termination 9.4

Mark
[see: type_mark]

Matching components
of arrays 4.5.2; 4.5.1, 5.2.1
of records 4.5.2

Mathematically correct result of a numeric operation 4.5; 4.5.7

MAX_INT
[see: system.max_int]

Maximum line length 14.3

Maximum page length 14.3

Meaning
of a based literal 2.4.2
of a decimal literal 2.4.1
of a designator 6.7
of an identifier 6.6, 6.7, 8.3 [see

also: overloading of..., visibility]
of a literal 4.2

Membership test 4.4, 4.5.2
cannot be overloaded 6.7
in a relation 4.4

Membership test operation 4.5
[see also: overloading of...]
as a basic operation 3.3.3; 3.3, 3.5.5, 3.5.8, 3.5.10, 3.6.2, 3.7.4, 3.8.2, 7.4.2

MEMORY_SIZE (predefined named number)
[see: system.memory_size]

MIN_INT
[see: system.min_int]

Minus
character [see: hyphen character]
character in an exponent of a numeric literal 2.4.1
delimiter 2.2
operator [see: binary adding operator, unary adding operator]
unary operation 4.5.4

Mod operator
[see: multiplying operator]

MODE (input-output function)
in text_io 14.2.1; 14.3.3, 14.3.4, 14.3.10

Mode (of a file) 14.1; 14.2.1
of a text_io file 14.3.1; 14.3.4

Mode (of a parameter) 6.1; D
[see also: formal parameter]
of a formal parameter of a renamed subprogram 8.5
of a parameter 12.1.1

Mode in out for a formal parameter 6.1, 6.2; 3.2.1
of a function is not allowed 6.5

Mode in for a formal parameter 6.1, 6.2; 3.2.1
of a function 6.5

Mode out for a formal parameter 6.1, 6.2
of a function is not allowed 6.5

MODE_ERROR (input-output exception) 14.4; 14.2.2, 14.2.3, 14.2.4, 14.2.5, 14.3.1, 14.3.3, 14.3.4, 14.3.5, 14.3.10, 14.5

Modulus operation 4.5.5

Multidimensional array 3.6

Multiple
component definition 3.7
object declaration 3.2

Multiplication operation 4.5.5

Multiplying operator 4.5; 4.5.5, C
[see also: arithmetic operator, overloading of an operator]
in a term 4.4
overloaded 6.7

Must (legality requirement) 1.6

Mutually recursive types 3.8.1; 3.3.1

Name of
an attribute 4.1.4
a base type 3.3; 12.1.1
a block 5.6
a character 2.1
a component of an array 4.1.1
a component of a record 3.7, 4.1.3
a component system dependent record component 13.4
a delimiter 2.2
an exception 11.1; 11
an exception in an exception choice 11.2
an exception in a raise statement 11.3
an exception in a renaming declaration 8.5
a formal parameter 6.1
a formal parameter and overload resolution 6.6
a function [see: designator]
a function in a function call 6.4
a function which is the current instantiation 12.1
a library unit 10.1, 10.1.1
a loop 5.5
a numeric value 3.2
an object 3.2
an object as a primary 4.4
an object in a renaming declaration 8.5
an object starting with a prefix 4.1.3
an operator [see: designator]
a package 7.1
a package in a renaming declaration 8.5
a package in a use clause 8.4; 10.1.1
a parameter of a predefined operator 4.5
a private type 7.4.1; 3.3
a procedure in a procedure call statement 6.4
a procedure which is the current instantiation 12.1
a program unit 8.6
a subcomponent of an unconstrained variable 12.3.1
a subprogram 6.1; 6.3.2, 12.1.3 [see also: designator]
a subprogram in a renaming declaration 8.5
a subtype 3.3.1, 3.3.2; 12.1.1, 12.1.3
[see also: type_mark of a subtype, type_mark]
a type 3.3.1; 3.3.2, 12.1.1, 12.1.3
[see also: type_mark of a type, type_mark]
a variable 5.2
a variable as an actual parameter 6.4, 6.4.1

NAME (input-output function)
in text_io 14.2.1

NAME (predefined type)
[see: system.name]

Name (of an entity) 4.1; 2.3, 3.1, D
[see also: attribute, block name, denote, evaluation of..., function call, identifier, indexed component, label, loop name, loop parameter, loop simple name, operator symbol, renaming declaration, selected component, simple name, type_mark, visibility]
as a prefix 4.1
as a primary 4.4
as the expression in a case statement 5.4
declared by renaming is not allowed
as prefix of certain expanded names 4.1.3
denoting an entity 4.1
starting with a prefix 4.1; 4.1.1, 4.1.3, 4.1.4

Name string (of a file) 14.1; 14.2.1, 14.2.3, 14.2.5, 14.3, 14.3.10, 14.4

NAME_ERROR (input-output exception) 14.4; 14.2.1, 14.2.3, 14.2.5, 14.3.10, 14.5

Named association 6.4.2, D
[see also: component association, parameter association]

Named block statement
[see: block name]

Named loop statement
[see: loop name]

Named number 3.2; 3.2.2
as an entity 3.1
as a primary 4.4
in a static expression 4.9

NATURAL (predefined integer subtype) C; 14.3.6

Negation operation 4.5.4

Negative exponent
in a numeric literal 2.4.1
to an exponentiation operator 4.5.6

NEW_LINE (text_io procedure) 14.3.4; 14.3.5, 14.3.6, 14.3.10
raising an exception 14.4

NEW_PAGE (text_io procedure) 14.3.4; 14.3.10
raising an exception 14.4

No other effect of elaboration 3.9

Not defined by the language
[see: incorrect order dependence]

Not equal
compound delimiter [see: inequality compound delimiter]
operator [see: relational operator]

Not in membership test
[see: membership test]

Not unary operation 4.5.6

Not unary operator
[see: highest precedence operator]
as an operation of an array type 3.6.2
as an operation of boolean type 3.5.5
in a factor 4.4

Not yet elaborated 3.9

Null array 3.6.1; 3.6
aggregate 4.3.2
and relational operation 4.5.2
as an operand of a catenation 4.5.3

Null component list 3.7

Null literal 3.8, 4.2
[see also: overloading of...]
as a basic operation 3.3.3; 3.8.2
as a primary 4.4
may not be the argument of a conversion 4.6

Null range 3.5
for a loop parameter 5.5

Null record 3.7
and relational operation 4.5.2

Null statement 5.1
[see also: statement]
as a simple statement 5.1

Null string literal 2.6

Number
[see: based number, decimal number]

Number declaration 3.2, 3.2.2
as a basic declaration 3.1

NUMBER_BASE (predefined integer subtype) 14.3.7; 14.3.10

Numeric literal 2.4, 4.2; 2.2, 2.4.1, 2.4.2
[see also: universal type expression]
and an adjacent separator 2.2
as a basic operation 3.3.3
as a primary 4.4
as the parameter of value attribute 3.5.5
as the result of image attribute 3.5.5
assigned 5.2
can be written in the basic character set 2.10
in a conforming construct 6.3.1
in a static expression 4.9

Numeric operation of a universal type 4.10

Numeric type 3.5
[see also: conversion, integer type, scalar type]
operation 4.5, 4.5.2, 4.5.3, 4.5.4, 4.5.5

Numeric value of a number declaration 3.2

NUMERIC_ERROR (predefined exception) 11.1
[see also: division check, overflow check]
not raised due to lost overflow conditions 13.7.3
raised by a numeric operator 4.5
raised by a predefined integer operation 3.5.4
raised by a universal expression 4.10
raised by integer division remainder or modulus 4.5.5
raised due to a conversion out of range 3.5.4, 3.5.6
suppressed 11.7

Object 3.2; 3.2.1, D
[see also: collection, component, constant, formal parameter, initial value, loop parameter, name of..., size attribute, subcomponent, variable]
as an actual parameter 6.2
as an entity 3.1
created by elaboration of an object declaration 3.2.1
of a file type [see: file]
renamed 8.5

Object declaration 3.2, 3.2.1
[see also: elaboration]
as a basic declaration 3.1
as a full declaration 7.4.3
in a package specification 7.1
of an array object 3.6.1
of a record object 3.7.2
with a limited type 7.4.4

Object module
from another language system 13.9

Obsolete compilation unit (due to recompilation) 10.3

Occur immediately within
[see: immediately within]

Omitted parameter association for a subprogram call 6.4.2

Open file 14.1

Operation 3.3, 3.3.3; D
[see also: basic operation, direct visibility, operator, predefined operation, visibility by selection, visibility]
as an entity 3.1
classification 3.3.3
of an array type 3.6.2
of a discrete type 3.5.5
of a limited type 7.4.4
of a private type 7.4.2; 7.4.1
of a record type 3.7.4
of a subtype 3.3
of a subtype of a discrete type 3.5.5
of a type 3.3
of a universal type 4.10
propagating an exception 11.6

Operator 4.5; 4.4, C, D
[see also: binary adding operator, designator, exponentiating operator, function, highest precedence operator, logical operator, multiplying operator, overloading of..., predefined operator, relational operator, unary adding operator]
as an operation 3.3.3 [see also: operation]
implicitly declared 3.3.3
in an expression 4.4
in a static expression 4.9
overloaded 6.7; 6.6
renamed 8.5

Operator declaration 6.1; 4.5, 6.7

Operator symbol 6.1
[see also: direct visibility, overloading of .., scope of..., visibility by selection, visibility]
as a designator 6.1
as a designator in a function declaration 4.5
as a name 4.1
before arrow compound delimiter 8.3
declared 3.1
in a renaming declaration 8.5
in a selector 4.1.3
in a static expression 4.9
not allowed as the designator of a library unit 10.1
of homograph declarations 8.3
overloaded 6.7; 6.6

Optimization 10.6
and exceptions 11.6

Or else control form
[see: short circuit control form]

Or operator
[see: logical operator]

Order
[see: Lexicographic order]

Order not defined by the language
[see: incorrect order dependence]

Order of application of operators in an expression 4.5

Order of compilation (of compilation units) 10.1, 10.3; 10.1.1, 10.4; (of compilation units) creating recompilation dependence 10.3

Order of copying back of out and in out formal parameters 6.4

Order of elaboration (of compilation units) 10.5; 10.1.1

Order of evaluation 1.6
[see also: erroneous]
and exceptions 11.6
of conditions in an if statement 5.3

of expressions and the name in an assignment statement 5.2
of operands in an expression 4.5
of parameter associations in a subroutine call 6.4
of the bounds of a range 3.5

Ordering operator 4.5; 4.5.2

Ordering relation 4.5.2
[see also: relational operator]
of a scalar type 3.5

Other effect
[see: no other effect]

Other special character 2.1
[see also: graphic character]

Others 3.7.3
as a choice in an array aggregate 4.3.2
as a choice in a case statement alternative 5.4
as a choice in a component association 4.3
as a choice in a record aggregate 4.3.1
as an exception choice 11.2

Out mode
[see: mode out]

OUT_FILE (input-output file_mode enumeration literal) 14.1

Overflow_check
[see: numeric_error, suppress]

Overlapping scopes
[see: hiding, overloading]

Overlaying of objects or program units 13.5

Overloading 8.3; D
[see also: designator, homograph declaration, identifier, operator symbol, scope, simple name, subprogram, visibility];
in an assignment statement 5.2
in an expression 4.4
resolution 6.6
resolution context 8.7
resolved by explicit qualification 4.7
sources of 6.6, 8.3
subject to visibility 8.3

Overloading of
an aggregate 4.3; 3.4
a declaration 8.3
a designator 6.6; 6.7
an enumeration literal 3.5.1; 3.4
an explicitly converted expression 4.6
an identifier 6.6
an implicitly converted expression 4.6
a library unit by a locally declared subprogram 10.1
a library unit by means of renaming 10.1
a literal 4.2
a membership test 4.5.2
a null literal 4.2; 3.8
an operator 4.5, 6.7; 4.4, 6.1
an operator symbol 6.6; 6.7
a string literal 4.2
a subprogram 6.6; 6.7
the expression in a case statement 5.4

Package 7, 7.1; D
[see also: deferred constant declaration, library unit, name of..., predefined package, private part, program unit, visible part]
as an entity 3.1
including a raise statement 11.3
named in a use clause 8.4
renamed 8.5
with a separately compiled body 10.2

Package body 7.1, 7.3; D
as a proper body 3.9
as a secondary unit 10.1
as a secondary unit compiled after the corresponding library unit 10.3
in another package body 7.1
including an exception handler 11.2; 11
including an exit statement 5.7
including an implicit declaration 5.1
must be in the same declarative part as the declaration 3.9
raising an exception 11.4.1, 11.4.2
recompilation does not affect other compilation units 10.3
recompiled 10.3

Package declaration 7.1, 7.2; D
and body as a declarative region 8.1
as a basic declaration 3.1
as a later declarative item 3.9
as a library unit 10.1
determining the visibility of another declaration 8.3
elaboration raising an exception 11.4.2
in a package specification 7.1
must be in the same declarative part as the body 3.9
recompiled 10.3

Package identifier 7.1

Package specification 7.1, 7.2

Page 14.3.4; 14.3

PAGE (text_io function) 14.3.4; 14.3.10
raising an exception 14.4

Page length 14.3.3; 14.3, 14.3.1, 14.3.4, 14.4

Page terminator 14.3; 14.3.3, 14.3.4, 14.3.5

PAGE_LENGTH (text_io function) 14.3.3, 14.3.4; 14.3.10
raising an exception 14.4

Paragraphing recommended for the layout
of programs 1.5

Parameter D
[see also: actual parameter,
formal parameter, formal part, function,
loop parameter, mode, procedure,
subprogram]
of a main program 10.1

Parameter and result type profile 6.6; 8.5
of homograph declarations 8.3

Parameter association 6.4, 6.4.1
named parameter association 6.4
named parameter association for selective
visibility 8.3
omitted for a subprogram call 6.4.2
positional parameter association 6.4

Parameter declaration
[see: parameter specification]

Parameter part
[see: actual parameter part]

Parameter specification 6.1
[see also: loop parameter specification]
as part of a basic declaration 3.1
having an extended scope 8.2
in a formal part 6.1
visibility 8.3

Parameter type profile 6.6

Parenthesis
character 2.1
delimiter 2.2

Parenthesized expression
as a primary 4.4; 4.5
in a static expression 4.9

Part
[see: actual parameter part, declarative
part, formal part]

Partial ordering of compilation 10.3

Percent character 2.1
[see also: string literal]
replacing quotation character 2.10

Period character 2.1
[see also: dot character, point character]

Physical processor 9; 9.8

Plus
character 2.1
delimiter 2.2
operator [see: binary adding operator,
unary adding operator]
unary operation 4.5.4

Point character 2.1
[see also: dot]
in a based literal 2.4.2
in a decimal literal 2.4.1
in a numeric literal 2.4

Point delimiter 2.2

Point in the program text 8.3

Pool of logically related entities
[see: package]

POS (predefined attribute) 3.5.5; A, 13.3

Position number
as parameter to val attribute 3.5.5
of an enumeration literal 3.5.1
of an integer value 3.5.4
of a value of a discrete type 3.5
returned by pos attribute 3.5.5

Positional association 6.4.2, D
[see also: component association, parameter
association]

POSITIVE (predefined integer subtype) 3.6.3;
C, 14.3.7, 14.3.8, 14.3.9, 14.3.10
as the index type of the string type 3.6.3

POSITIVE_COUNT (predefined integer subtype)
14.2.5, 14.3.10; 14.2.4, 14.3, 14.3.4

Potentially affected compilation unit
(due to a recompilation) 10.3

Potentially visible declaration 8.4

Pound sterling character 2.1

Power operator
[see: exponentiating operator]

Precedence 4.5

PRED (predefined attribute) 3.5.5; A, 13.3

Predecessor
[see: pred attribute]

Predefined attribute
[see: base, callable, constrained,
count, first, image, last,
pos, pred, range, size, small,
succ, terminated, val, value, width]

Predefined constant 8.6; C
[see also: system.system_name]
for CHARACTER values [see: ascii]

Predefined exception 8.6, 11.1; C, 11.4.1
[see also: constraint_error, io_common,
numeric_error, program_error]

Predefined function 8.6; C
[see also: attribute, character literal, enumeration literal]

Predefined identifier 8.6; C

Predefined library package 8.6; C
[see also: predefined package, ascii, input-output, package, system, text_io]

Predefined named number
[see: system.max_int, system.min_int]

Predefined operation 3.3, 3.3.3; 8.6
[see also: operation, predefined operator]
of a discrete type 3.5.5
of a universal type 4.10
propagating an exception 11.6

Predefined operator 4.5, 8.6; C
[see also: abs, arithmetic operator, binary adding operator, catenation, equality, exponentiating operator, highest precedence operator, inequality, limited type, logical operator, multiplying operator, operator, predefined operation, relational operator, unary adding operator]
applied to an undefined value 3.2.1
as an operation 3.3.3
for an array type 3.6.2
for a record type 3.7.4
implicitly declared 3.3.3
in a static expression 4.9
of an integer type 3.5.4
raising an exception 11.4.1

Predefined package 8.6; C
[see also: ascii, input-output package, library unit, predefined library package, standard]
for input-output 14

Predefined subprogram 8.6; C
[see also: input-output subprogram, library unit]

Predefined subtype 8.6; C
[see also: field, natural, number_base, positive, priority]

Predefined type 8.6; C
[see also: boolean, character, count, integer, string, system.name, universal_integer]

Prefix 4.1; D
[see also: appropriate for a type, function call, name, selected component, selector]
in an attribute 4.1.4
in an indexed component 4.1.1
in a selected component 4.1.3
which is a function call 4.1
which is a name 4.1

Primary 4.4
in a factor 4.4
in a static expression 4.9

Private part (of a package) 7.2; 7.4.1, 7.4.3, D
[see also: deferred constant declaration, private type declaration]

Private type 3.3, 7.4, 7.4.1; D
[see also: class of type, limited private type, type with discriminants]
corresponding full type declaration 3.3.1
formal parameter 6.2
of a deferred constant 7.4; 3.2.1
operation 7.4.2
with discriminants subject to an attribute 3.7.4

Private type declaration 7.4; 7.4.1, 7.4.2
[see also: private part (of a package), visible part (of a package)]
acting as an incomplete type declaration 3.8.1
as a portion of a declarative region 8.1
determining the visibility of another declaration 8.3
including the word ’limited’ 7.4.4

Procedure 6.1; 6
[see also: name of a procedure, parameter and result type profile, parameter, subprogram]
as a main program 10.1
renamed 8.5

Procedure body
[see: subprogram body]

Procedure call 6.4; 6, D
[see also: subprogram call]

Procedure call statement 6.4
[see also: actual parameter, statement]
as a simple statement 5.1

Procedure specification
[see: subprogram specification]

Processor 9

Profile
[see: parameter and result type profile, parameter type profile]

Program 10; D
[see also: main program]

Program legality 1.6

Program library 10.1, 10.4; 10.5
creation 10.4; 13.7
manipulation and status 10.4

Program text 2.2, 10.1; 2.10

Program unit 6, 7, 9, 12; D
[see also: library unit, package, subprogram]
entity denoted by an expanded name 4.1.3

with a separately compiled body 10.2

PROGRAM_ERROR (predefined exception) 11.1
[see also: elaboration_check, erroneousness, storage_check]
raised by an erroneous program or incorrect order dependence 1.6; 11.1
raised by a subprogram call before elaboration of the body 3.9; 7.3
raised by reaching the end of a function body 6.5
suppressed 11.7

Propagation of an exception
[see: exception propogation]

Proper body 3.9
as a body 3.9
of a library unit separately compiled 10.1

PUT (text_io procedure) 14.3.5; 14.3, 14.3.2, 14.3.10
for character and string types 14.3.6
for enumeration types 14.3.9
for integer types 14.3.7
raising an exception 14.4

Qualification operation 4.7
as a basic operation 3.3.3; 3.3, 3.5.5, 3.6.2, 3.7.4, 3.8.2, 7.4.2
as a basic operation, 3.5.8
using a name of an enumeration type as qualifier 3.5.1

Qualified expression 4.7; D
as a primary 4.4
in a case statement 5.4
in a static expression 4.9
qualification of an array aggregate 4.3.2
to resolve an overloading ambiguity 6.6

Quotation character 2.1
in a string literal 2.6
replacement by percent character 2.10

Raise statement 11.3; 11
[see also: exception, statement]
as a simple statement 5.1
including the name of an exception 11.1

Raising of an exception 11, 11.3; D
[see also: exception]
causing a transfer of control 5.1

Range 3.5; D
[see also: discrete range, null range]
as a discrete range 3.6
in a relation 4.4
of an index subtype 3.6
of an integer type containing the result of an operation 4.5
of a predefined integer type 3.5.4
yielded by an attribute 4.1.4

RANGE (predefined attribute) 3.5; 4.1.4, A
for an array type, 3.6.2

Range constraint 3.5; D
[see also: elaboration of...]
ignored due to range_check suppression 11.7
in an integer type definition 3.5.4
in a subtype indication 3.5; 3.3.2
violated 11.1

Range_check
[see: constraint_error, suppress]

Reading the value of an object 6.2

Reciprocal operation in exponentiation by a negative integer 4.5.6

Recompilation 10.3

Record aggregate 4.3.1; 4.3, D
[see also: aggregate]
in a code statement 13.8

Record component
[see: component, record type, selected component]

Record type 3.7; 3.3, D
[see also: component, composite type matching components, subcomponent]
formal parameter 6.2
including a limited subcomponent 7.4.4
operation 3.7.4

Record type declaration
[see: record type definition, type declaration]
as a declarative region 8.1
determining the visibility of another declaration 8.3

Record type definition 3.7; 3.3.1
[see also: component declaration]

Recursive
call of a subprogram 6.1, 12.1; 6.3.2
types 3.8.1; 3.3.1

Reentrant subprogram 6.1

Reference parameter passing 6.2

Relation 4.4
in an expression 4.4

Relational expression
[see: relation, relational operator]

Relational operation 4.5.2
of a boolean type 3.5.3
of a discrete type 3.5.5
of a scalar type 3.5

Relational operator 4.5; 4.5.2, C
[see also: equality operator, inequality operator, ordering relation, overloading of an operator, predefined operator]
for an array type 3.6.2

for a private type 7.4.2
for a record type 3.7.4
in a relation 4.4
overloaded 6.7

Rem operator
[see: multiplying operator]

Remainder operation 4.5.5

Renaming declaration 8.5; 4.1, 12.1.3
[see also: name]
as a basic declaration 3.1
as a declarative region 8.1
for an array object 3.6.1
for a record object 3.7.2
name declared is not allowed as a prefix of certain expanded names 4.1.3
to overload a library unit 10.1
to resolve an overloading ambiguity 6.6

Replacement of characters in program text 2.10

Reserved word 2.9; 2.2, 2.3
must not be declared 2.9

Resolution of overloading
[see: overloading]

Result of a function
[see: returned value]

Result subtype (of a function) 6.1
of a return expression 5.8

Result type and overload resolution 6.6

Result type profile
[see: parameter and...]

Return
[see: carriage return]

Return statement 5.8
[see also: function, statement]
as a simple statement 5.1
causing a loop to be exited 5.5
causing a transfer of control 5.1
completing block statement execution 9.4
completing subprogram execution 9.4
expression which is an array aggregate 4.3.2
in a function body 6.5

Returned value
[see: function call]
of a function call 5.8, 6.5; 8.5
of a main program 10.1
of an operation 3.3.3
of a predefined operator of an integer type 3.5.4

Right label bracket compound delimiter 2.2

Right parenthesis
character 2.1
delimiter 2.2

Run time check 11.7; 11.1

Satisfy (a constraint) 3.3; D
[see also: constraint, subtype]
an index constraint 3.6.1
a range constraint 3.5

Scalar type 3.3, 3.5; D
[see also: class of type, discrete type, enumeration type, integer type, numeric type, static Expression]
formal parameter 6.2
of a range in a membership test 4.5.2
operation 3.5.5; 4.5.2

Scheme
[see: iteration scheme]

Scope 8.2; 8.3, D
[see also: basic operation, character literal, declaration, declarative region, identifier, immediate scope, implicit declaration, operator symbol, overloading, visibility]
of a use clause 8.4

Secondary unit 10.1
[see also: compilation unit, library unit]

Selected component 4.1.3; 8.3, D
[see also: direct visibility, prefix, selector, visibility by selection, visibility]
as a basic operation 3.3.3; 3.3, 3.7.4, 3.8.2, 7.4.2
as a name 4.1
for selective visibility 8.3
in a conforming construct 6.3.1
starting with standard 8.6
using a block name 5.6
using a loop name 5.5
whose prefix denotes a package 8.3
whose prefix denotes a record object 8.3

Selection of an exception handler 11.4, 11.4.1, 11.4.2; 11.6

Selective visibility
[see: visibility by selection]

Selector 4.1.3; D
[see also: prefix, selected component]

Semicolon character 2.1

Semicolon delimiter 2.2

Separate compilation 10, 10.1; 10.5
of a proper body 3.9
of a proper body declared in another compilation unit 10.2

Separator 2.2

Sequence of statements 5.1
in a basic loop 5.5
in a block statement 5.6; 9.4
in a case statement alternative 5.4
in an exception handler 11.2
in an if statement 5.3
in a package body 7.1; 7.3
in a subprogram body 6.3; 9.4, 13.8
including a raise statement 11.3
raising an exception 11.4.1

Sequential access file 14.2; 14.1, 14.2.1

Sequential execution
[see: sequence of statements, statement]

Sequential input-output 14.2.2; 14.2.1

SET_COL (text_io procedure) 14.3.4; 14.3.10

SET_INPUT (text_io procedure) 14.3.2; 14.3.10
raising an exception 14.4

SET_LINE (text_io procedure) 14.3.4; 14.3.10

SET_LINE_LENGTH
(text_io procedure) 14.3.3; 14.3.10
raising an exception 14.4

SET_OUTPUT (text_io procedure) 14.3.2; 14.3.10
raising an exception 14.4

SET_PAGE_LENGTH
(text_io procedure) 14.3.3; 14.3.10
raising an exception 14.4

Sharp character 2.1
[see also: based literal]
replacement by colon character 2.10

Short circuit control form 4.5, 4.5.1; 4.4
as a basic operation 3.3.3; 3.5.5
in an expression 4.4

Short-circuit control form cannot be overloaded 6.7

Simple expression 4.4
as a choice 3.7.3
as a choice in an aggregate 4.3
as a range bound 3.5
in a relation 4.4

Simple name 4.1; 2.3, D
[see also: block name, identifier, label,
loop name, loop simple name, name of...,
name, overloading, visibility]
as a choice 3.7.3
as an enumeration literal 3.5.1
as a formal parameter 6.4
as a name 4.1
before arrow compound delimiter 8.3
in an attribute designator 4.1.4
in a conforming construct 6.3.1
in a package body 7.1
in a package specification 7.1
in a selector 4.1.3
in a with clause 10.1.1
versus identifier 3.1

Simple name of...
[see: name of...]

Simple statement 5.1
[see also: statement]

Single
component definition 3.7
object declaration 3.2

SKIP_LINE (text_io procedure) 14.3.4; 14.3.10
raising an exception 14.4

SKIP_PAGE (text_io procedure) 14.3.4; 14.3.10
raising an exception 14.4

Some order not defined by the language
[see: incorrect order dependence]

Space character 2.1
[see also: basic graphic character]
as a separator 2.2
in a comment 2.7
not allowed in an identifier 2.3
not allowed in a numeric literal 2.4.1

Space character literal 2.5; 2.2

Special character 2.1
[see also: basic graphic character, other
special character]
in a delimiter 2.2

Specification
[see: declaration,
enumeration literal specification,
loop parameter specification,
package specification, parameter specification,
subprogram specification]

STANDARD (predefined package) 8.6; C
[see also: library unit]
as a declarative region 8.1
enclosing the library units of a program
10.1.1; 10.1, 10.2

STANDARD_INPUT
(text_io function) 14.3.2; 14.3.10

STANDARD_OUTPUT
(text_io function) 14.3.2; 14.3.10

Star
[see: double star]
character 2.1
delimiter 2.2

Statement 5.1; 5, D
[see also: abort statement

assignment statement,
block statement, case statement,
compound statement, exit statement,
if statement, label, loop statement, null
statement, procedure call statement, raise
statement, return statement,
sequence of statements, simple statement,
target statement]
allowed in an exception handler 11.2
as an entity 3.1
as an overload resolution context 8.7
optimized 10.6
raising an exception 11.4.1; 11.4
that cannot be reached 10.6

Statement alternative
[see: case statement alternative]

Static constraint 4.9
on a subcomponent subject to a component
clause 13.4
on a type 3.5.4, 3.5.6, 3.5.9, 13.2

Static discrete range 4.9
as a choice of an aggregate 4.3.2
as a choice of a case statement 5.4

Static expression 4.9; 8.7, D
as a bound in an integer type definition 3.5.4
as a choice in a case statement 5.4
for a choice in a record aggregate 4.3.2
in an attribute designator 4.1.4
in a number declaration 3.2, 3.2.2
which is of universal type 4.10

Static others choice 4.3.2

Static subtype 4.9
of the expression in a case statement 5.4

STATUS_ERROR (input-output exception) 14.4;
14.2.1, 14.2.2, 14.2.3, 14.2.4, 14.2.5, 14.3.3,
14.3.4, 14.3.5, 14.3.10, 14.5

STORAGE_ERROR (predefined exception)
raised by an elaboration of a declarative item 11.1
raised by the execution of a subprogram call 11.1

STRING (predefined type) 3.6.3; C
[see also: predefined type]
as the parameter of value attribute 3.5.5
as the result of image attribute 3.5.5

String bracket 2.6; 2.10

String literal 2.6, 4.2; 2.2, 3.6.3
[see also: overloading of..., percent mark
character, quotation character]
as a basic operation 3.3.3, 4.2; 3.6.2
as an operator symbol 6.1
as a primary 4.4
may not be the argument of a conversion 4.6
replaced by a catenation of basic characters 2.10

Subaggregate 4.3.2

Subcomponent 3.3; D
[see also: component, composite type,
object]
of a component for which a component
clause is given 13.4
of an unconstrained variable 12.3.1
renamed 8.5
which is of a private type 7.4.1
which is of limited type 7.4.4

Subprogram 6; D
[see also: actual parameter, completed
subprogram, formal
parameter, function, library unit, name of
a subprogram, name of..., overloading of...,
parameter and result type profile, parameter,
predefined subprogram, procedure, program unit]
as an entity 3.1
as a main program 10.1
as an operation 3.3.3; 7.4.2
including a raise statement 11.3
overloaded 6.6
renamed 8.5
with a separately compiled body 10.2

Subprogram body 6.3; 6
as a library unit 10.1
as a proper body 3.9
as a secondary unit 10.1
as a secondary unit compiled after the
corresponding library unit 10.3
in a package body 7.1
including an exception handler 11.2; 11
including an exit statement 5.7
including an implicit declaration, 5.1
including a return statement 5.8
must be in the same declarative part
as the declaration 3.9
not yet elaborated at a call 3.9
raising an exception 11.4.1, 11.4.2
recompilation does not affect other
compilation units 10.3
recompiled 10.3

Subprogram call 6.4; 6, 6.3, 12.3
[see also: actual parameter, function call, procedure
call statement, procedure call]
before elaboration of the body 3.9
with no elaborated body 11.1

Subprogram declaration 6.1; 6
and body as a declarative region 8.1
as a basic declaration 3.1
as a later declarative item 3.9
as a library unit 10.1
as an overloaded declaration 8.3
implied by the body 6.3
in a package specification 7.1
made directly visible by a use clause 8.4
must be in the same declarative part
as the body 3.9
of an operator 6.7

recompiled 10.3

Subprogram specification 6.1
conforming to another 6.3.1
for a function 6.5
in a renaming declaration 8.5
in a subprogram body 6.3
including the name of a private type 7.4.1

Subtraction operation 4.5.3

Subtype 3.3, 3.3.2; D
[see also: attribute of..., base attribute, constrained subtype, constraint, first named subtype, name of..., operation of..., result subtype, satisfy, size attribute, static subtype, type, unconstrained subtype]
as an entity 3.1
declared by an integer type declaration 3.5.4
in a membership test 4.5.2
name [see: name of a subtype, type_mark of a subtype]
not considered in overload resolution 8.7
of an actual parameter 6.4.1
of an array type [see: constrained array type, index constraint]
of a component of an array 3.6
of a component of a record 3.7
of a constant in a static expression 4.9
of a formal parameter 6.4.1
of a formal parameter or result of a renamed subprogram 8.5
of an object [see: elaboration of...]
of a private type 7.4, 7.4.1
of a record type [see: constrained record type, discriminant constraint]
of a scalar type 3.5
of a variable 5.2

Subtype conversion 4.6
[see also: convertion operation, explicit conversion, implicit conversion, type conversion]
in an array assignment 5.2.1; 5.2
in a static expression 4.9

Subtype declaration 3.3.2; 3.1, 3.3.2
[see also: elaboration of...]
as a basic declaration 3.1
including the name of a private type 7.4.1

Subtype definition
[see: component subtype definition, index subtype definition]

Subtype indication 3.3.2
[see also: elaboration of...]
as a component subtype indication 3.7
as a discrete range 3.6
in an array type definition 3.6
in a component declaration 3.7
in a constrained array definition 3.6
in an object declaration 3.2, 3.2.1
in an unconstrained array definition 3.6
with a range constraint 3.5

SUCC (predefined attribute) 3.5.5; A, 13.3

Successor
[see: succ attribute]

Symbol
[see: operator symbol]

Syntactic category 1.5

Syntax notation 1.5

Syntax rule 1.5; E

SYSTEM (predefined library package) 13.7; C

System dependent F
attribute 13.4
constant 13.7
input output 14.1
named number 13.7, 13.7.1
record component 13.4
type 13.7

SYSTEM.MAX_INT
(predefined named number)
13.7.1; 3.5.4
exceeded by the value of a universal expression 4.10

SYSTEM.MIN_INT (predefined named number)
13.7.1; 3.5.4
greater than the value of a universal expression 4.10

SYSTEM.NAME (predefined type) 13.7

SYSTEM.SYSTEM_NAME (predefined constant) 13.7
[see also: system_name]

Tabulation
[see: horizontal tabulation, vertical tabulation]

Target type of a conversion 4.6

Term 4.4
in a simple expression 4.4

Terminator
[see: file terminator, line terminator, page terminator]

Text input-output 14.3; 14.2.1

Text of a program 2.2, 10.1

TEXT_IO (predefined input-output package)
14.3; C, 14, 14.1, 14.3.9, 14.3.10
exceptions 14.4; 14.5
specification 14.3.10

Times operator
[see: multiplying operator]

Transfer of control
[see: exception, exit statement, return statement]

Transitive closure of dependencies of compilation units 10.5

Transliteration
[see: replacement]

TRUE boolean enumeration literal 3.5.3; C

Type 3.3; D
[see also: appropriate for a type, array type, attribute of..., base attribute, base type, boolean type, character type, class of type, composite type, constrained type, discrete type, enumeration type, integer type, limited private type, limited type, name of..., numeric type, operation of..., predefined type, private type, record type, scalar type, size attribute, subtype, unconstrained subtype, unconstrained type, universal type]
as an entity 3.1
due to elaboration [see: elaboration]
name [see: name of a type, type_mark of a type]
of an actual parameter 6.4.1
of an aggregate 4.3.1, 4.3.2
of a case statement expression 5.4
of a condition 5.3
of a declared object 3.2, 3.2.1
of an expression 4.4
of a file 14.1
of an index 4.1.1
of a loop parameter 5.5
of a named number 3.2, 3.2.2
of a primary in an expression 4.4
of a universal expression 4.10
of a value 3.3; 3.2
renamed 8.5
yielded by an attribute 4.1.4

Type conversion 4.6
[see also: conversion operator, conversion, explicit conversion, subtype conversion, unchecked_conversion]
as an actual parameter 6.4, 6.4.1
as a primary 4.4
in a static expression 4.9

Type declaration 3.3.1; D
[see also: elaboration of..., incomplete type declaration, private type declaration]
as a basic declaration 3.1
as a full declaration 7.4.1
implicitly declaring operations 3.3.3
in a package specification 7.1
including the name of a private type 7.4.1
of an integer type 3.5.4
of a subtype 13.1

Type definition 3.3.1; D
[see also: array type definition, elaboration of..., enumeration type definition, integer type definition]

Type mark (denoting a type or subtype) 3.3.2
[see also: name of a subtype, name of a type]
in a conversion 4.6
in a deferred constant declaration 7.4
in an index subtype definition 3.6
in a parameter specification 6.1; 6.2
in a qualified expression 4.7
in a relation 4.4
in a renaming declaration 8.5
in a subprogram specification 6.1
of a static scalar subtype 4.9

Unary adding operator 4.4, 4.5, C; 4.5.4
[see also: arithmetic operator, overloading of an operator, predefined operator]
as an operation of a discrete type 3.5.5
in a simple expression 4.4
overloaded 6.7

Unary operator 4.5; 4.5.4, 4.5.6, C
[see also: highest precedence operator, unary adding operator]

Unconstrained array definition 3.6

Unconstrained array type 3.6
as an actual to a formal private type 12.3.2
formal parameter 6.2

Unconstrained subtype 3.3, 3.3.2
[see also: constrained subtype, constraint, subtype, type]

Unconstrained type 3.3; 3.2.1, 3.6, 3.6.1, 3.7, 3.7.2
formal parameter 6.2

Unconstrained variable 3.2.1, 3.3, 3.6, 3.7; 12.3.1
as a subcomponent [see: subcomponent]

Underline character 2.1
in a based literal 2.4.2
in a decimal literal 2.4.1
in an identifier 2.3

Unhandled exception 11.4.1

Unit
[see: compilation unit, library unit, program unit]

Universal expression 4.10
assigned 5.2
in an attribute designator 4.1.4
which is static 4.10

Universal type 4.10
[see also: conversion, implicit conversion]
expression [see: expression, numeric literal]

of a named number 3.2.2; 3.2
result of an attribute [see: attribute]

UNIVERSAL_INTEGER
(predefined type) 3.5.4, 4.10; C
[see also: integer literal]
argument to a conversion 3.3.3, 4.6
attribute 3.5.5, 13.7.1, 13.7.2, 13.7.3; 9.9
bounds of a discrete range 3.6.1
bounds of a loop parameter 5.5
converted to an integer type 3.5.5
of integer literals 2.4, 4.2
result of an operation 4.10; 4.5

Updating the value of an object 6.2

Upper bound
[see: bound, last attribute]

Upper case letter 2.1
[see also: basic graphic character]
A to F in a based literal 2.4.2
E in a decimal literal 2.4.1
in an identifier 2.3

Use clause (to achieve direct visibility) 8.4; 8.3, D
[see also: context clause]
as a basic declarative item 3.9
as a later declarative item 3.9
in a context clause of a compilation unit 10.1.1
inserted by the environment 10.4

USE_ERROR (input-output exception) 14.4; 14.2.1, 14.2.3, 14.2.5, 14.3.3, 14.3.10, 14.5

VAL (predefined attribute) 3.5.5; A

Value
[see: assignment, evaluation, expression, initial value, returned value, subtype, type]
in a constant 3.2.1; 3.2
in a variable 3.2.1, 5.2; 3.2
of an array type 3.6; 3.6.1
of a based literal 2.4.2
of a boolean type 3.5.3
of a character literal 2.5
of a character type 3.5.2; 2.5, 2.6
of a decimal literal 2.4.1
of a record type 3.7
of a string literal 2.6; 2.10
of a string type 3.6.3; 2.6
returned by a function call [see: returned value]

VALUE (predefined attribute) 3.5.5; A

Value set
of a subtype 3.3, 3.3.2
of a type 3.3

Value-return parameter passing 6.2

Variable 3.2.1; D
[see also: name of..., object, unconstrained...]
as an actual parameter 6.2
declared in a package body 7.3
formal parameter 6.2
in an assignment statement 5.2
name [see: name of a variable]
of an array type as destination of an assignment 5.2.1
of a private type 7.4.1
renamed 8.5

Variable declaration 3.2.1

Vertical bar character 2.1
replacement by exclamation character 2.10

Vertical bar delimiter 2.2

Vertical tabulation format effector 2.1

Violation of a constraint
[see: constraint_error exception]

Visibility 8.3; 8.2, D
[see also: basic operation, character literal, direct visibility, hiding, identifier, name, operation, operator symbol, overloading, scope of..., selected component, simple name]
and renaming 8.5
determining multiple meanings of an identifier 8.4, 8.7; 8.5
determining order of compilation 10.3
due to a use clause 8.4
of a library unit 8.6
of a library unit due to a with clause 10.1.1
of a name for an enumeration literal 3.5.1
of a name of an exception 11.2
of an operation declared in a package 7.4.2
of a renaming declaration 8.5
of a subprogram declared in a package 6.3
of declarations in a package body 7.3
of declarations in a package specification 7.2
of declarations in the package system 13.7
of the simple name of a named number 3.2
of the simple name of an object 3.2
of the simple name of a subtype 3.3.2
of the simple name of a type 3.3.1

Visibility by selection 8.3
[see also: basic operation, character literal, operation, operator symbol, selected component]

Visible part (of a package) 7.2; 3.2.1, 7.4, 7.4.1, 7.4.3, D
[see also: deferred constant declaration, private type declaration]
declaration having an extended scope 8.2
determining the visibility of a declaration 8.3
entity denoted by an expanded name 4.1.3
named in a use clause 8.4

While loop
[see: loop statement]

WIDTH (predefined attribute) 3.5.5; A

With clause 10.1.1; D
  [see also: context clause]
    determining order of compilation 10.3
    determining the implicit order of library units 8.6
    in a context clause of a compilation
      unit 10.1.1
    inserted by the environment 10.4
    leading to direct visibility 8.3

Writing an output file 14.1, 14.2.1, 14.3.4

Xor operator
  [see: logical operator]

& operator
  [see: binary adding operator, catenation operator]

***** operator
  [see: multiplying operator]

+ operator
  [see: binary adding operator, unary adding operator]

- operator
  [see: binary adding operator, unary adding operator]

/= operator
  [see: inequality operator, relational operator]

/ operator
  [see: multiplying operator]

<= operator
  [see: relational operator]

< operator
  [see: relational operator]

= operator
  [see: equality operator, relational operator]

>= operator
  [see: relational operator]

> operator
  [see: relational operator]

Appendix H
Postscript : Submission of Comments

This appendix is not part of the standard definition of the AVA *programming language.*

We would appreciate comments on this AVA reference manual to be submitted by Internet to the address

mksmith@cli.com

If you do not have Internet access, please send the comments by mail

Michael K. Smith
Computational Logic Inc.
1717 W 6th, Suite 290
Austin, TX 78703

All comments may someday be sorted and processed mechanically in order to simplify their analysis and to facilitate giving them proper consideration. To aid this process, you are requested to precede each comment with a three-line header:

```
!section ...
!version Technical Report 64
!date  ...
!topic ...
```

The section line includes the section number, the paragraph number enclosed in parentheses, your name or affiliation (or both), and the date in ISO standard form (year-month-day). Paragraph numbers in section identifiers are optional, but very helpful. For example, here is the section line of comment #1194 on a previous version of the Ada Manual:

```
!section 03.02.01(12) D. Taffs 82-04-26
```

The version line, for comments on this document, should only contain "!version Technical Report 64". Its purpose is to distinguish comments that refer to different versions of this report.

The topic line should contain a one-line summary of the comment. This line is essential, and you are kindly asked to avoid topics such as "Typo" or "Editorial comment" which will not convey any information when printed in a table of contents. An example of an informative topic line is:

```
!topic Subcomponents of constants are constants
```

Note also that nothing prevents the topic line from including all the information of a comment, as in the following topic line:

```
!topic Insert:  "... are {implicitly} defined by a subtype declaration"
```

As a final example here is a complete comment received on a version of the Ada manual:

!section 03.02.01(12) D. Taffs 82-04-26
!version 10
!topic Subcomponents of constants are constants

Change "component" to "subcomponent" in the last sentence.

Otherwise the statement is inconsistent with the defined use of subcomponent in 3.3, which says that subcomponents are excluded when the term component is used instead of subcomponent.

References

[ARTEWG 87] *Catalogue of Ada Runtime Implementation Dependencies*
ACM, Special Interest Group on Ada, Runtime Environment Working Group, 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.

[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.

[NYU 83] Ada Project.
Ada/Ed Semantic Actions (Version 1.1).
Technical Report, Courant Institute, New York University, 1983.

[NYU 84] Ada Project.
Executable Semantic Model for Ada (Version 1.4).
Technical Report, Courant Institute, New York University, 1984.

[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.

[SmithMK 88] Michael K. Smith, Dan Craigen, and Mark Saaltink.
The nanoAVA Definition.
Technical Report 21, Computational Logic, Inc., June, 1988.

[SmithMK 90] Michael K. Smith.
A Formal Static Semantics for Part of AVA: ACL2 Version.
Technical Report 191, Computational Logic, Inc., May, 1990.

[Wichmann 89a] Wichmann, B. A.
Low-Ada: An Ada validation Tool.
NPL Report DITC 144/89, National Physical Laboratory, Teddington, Middlesex, United Kingdom, August, 1989.

[Wichmann 89b] Wichmann, B. A.
Insecurities in the Ada Programming Language.
NPL Report DITC 137/89, National Physical Laboratory, Teddington, Middlesex, United Kingdom, January, 1989.

Table of Contents

Chapter 1. Introduction 3

1.1. Scope of the Standard 3
1.1.1. Extent of the Standard 3
1.1.2. Conformity of an Implementation With the Standard 4
1.2. Structure of the Standard 4
1.3. Design Goals and Sources: Omitted 5
1.4. Language Summary 5
1.5. Method of Description and Syntax Notation 7
1.6. Classification of Errors 8

Chapter 2. Lexical Elements 11

2.1. Character Set 11
2.2. Lexical Elements, Separators, and Delimiters 13
2.3. Identifiers 14
2.4. Numeric Literals 14
2.4.1. Decimal Literals 15
2.4.2. Based Literals 15
2.5. Character Literals 16
2.6. String Literals 16
2.7. Comments 17
2.8. Pragmas: Removed 18
2.9. Reserved Words 18
2.10. Allowable Replacements of Characters 19

Chapter 3. Declarations and Types 21

3.1. Declarations 21
3.2. Objects and Named Numbers 22
3.2.1. Object Declarations 23
3.2.2. Number Declarations 25
3.3. Types and Subtypes 25
3.3.1. Type Declarations 26
3.3.2. Subtype Declarations 27
3.3.3. Classification of Operations 28
3.4. Derived Types : Removed 30
3.5. Scalar Types 30
3.5.1. Enumeration Types 31
3.5.2. Character Types 33
3.5.3. Boolean Types 33
3.5.4. Integer Types 33
3.5.5. Operations of Discrete Types 35
3.5.6. Real Types: Removed 37
3.5.7. Floating Point Types: Removed 37
3.5.8. Operations of Floating Point Types: Removed 37

3.5.9. Fixed Point Types: Removed . . . 38
3.5.10. Operations of Fixed Point Types: Removed . . . 38
3.6. Array Types . . . 38
3.6.1. Index Constraints and Discrete Ranges . . . 40
3.6.2. Operations of Array Types . . . 42
3.6.3. The Type String . . . 43
3.7. Record Types . . . 44
3.7.1. Discriminants: Removed . . . 45
3.7.2. Discriminant Constraints: Removed . . . 46
3.7.3. Variant Parts: Removed . . . 46
3.7.4. Operations of Record Types . . . 46
3.8. Access Types: Removed . . . 46
3.9. Declarative Parts . . . 47

Chapter 4. Names and Expressions . . . 49

4.1. Names . . . 49
4.1.1. Indexed Components . . . 50
4.1.2. Slices: Removed . . . 51
4.1.3. Selected Components . . . 51
4.1.4. Attributes . . . 52
4.2. Literals . . . 53
4.3. Aggregates . . . 54
4.3.1. Record Aggregates . . . 55
4.3.2. Array Aggregates . . . 56
4.4. Expressions . . . 58
4.5. Operators and Expression Evaluation . . . 59
4.5.1. Logical Operators and Short-circuit Control Forms . . . 61
4.5.2. Relational Operators and Membership Tests . . . 62
4.5.3. Binary Adding Operators . . . 63
4.5.4. Unary Adding Operators . . . 64
4.5.5. Multiplying Operators . . . 65
4.5.6. Highest Precedence Operators . . . 66
4.5.7. Accuracy of Operations with Real Operands: Removed . . . 67
4.6. Type Conversions . . . 67
4.7. Qualified Expressions . . . 69
4.8. Allocators: Removed . . . 70
4.9. Static Expressions and Static Subtypes . . . 71
4.10. Universal Expressions . . . 72

Chapter 5. Statements . . . 75

5.1. Simple and Compound Statements - Sequences of Statements . . . 75
5.2. Assignment Statement . . . 76
5.2.1. Array Assignments . . . 77
5.3. If Statements . . . 78
5.4. Case Statements . . . 79
5.5. Loop Statements . . . 80
5.6. Block Statements . . . 82
5.7. Exit Statements . . . 83
5.8. Return Statements . . . 83
5.9. Goto Statements: Removed . . . 84

Chapter 6. Subprograms 85

6.1. Subprogram Declarations 85
6.2. Formal Parameter Modes 86
6.3. Subprogram Bodies 87
6.3.1. Conformance Rules 89
6.3.2. Inline Expansion of Subprograms: Removed 90
6.4. Subprogram Calls 90
6.4.1. Parameter Associations 91
6.4.2. Default Parameters: Removed 93
6.5. Function Subprograms 93
6.6. Parameter and Result Type Profile - Overloading of Subprograms 94
6.7. Overloading of Operators: Removed 95

Chapter 7. Packages 97

7.1. Package Structure 97
7.2. Package Specifications and Declarations 98
7.3. Package Bodies 99
7.4. Private Type and Deferred Constant Declarations 100
7.4.1. Private Types 101
7.4.2. Operations of a Private Type 102
7.4.3. Deferred Constants 104
7.4.4. Limited Types: Removed. 105
7.5. Example of a Table Management Package 105
7.6. Example of a Text Handling Package: Removed 106

Chapter 8. Visibility Rules 107

8.1. Declarative Region 107
8.2. Scope of Declarations 108
8.3. Visibility 109
8.4. Use Clauses 112
8.5. Renaming Declarations 114
8.6. The Package Standard 116
8.7. The Context of Overload Resolution 117

Chapter 9. Tasks: Removed 119

Chapter 10. Program Structure and Compilation Issues 121

10.1. Compilation Units - Library Units 121
10.1.1. Context Clauses - With Clauses 123
10.1.2. Examples of Compilation Units 125
10.2. Subunits of Compilation Units : Removed 127
10.3. Order of Compilation 127
10.4. The Program Library 129
10.5. Elaboration of Library Units 129
10.6. Program Optimization 130

Chapter 11. Exceptions 133

11.1. Exception Declarations 133
11.2. Exception Handlers 134
11.3. Raise Statements 135
11.4. Exception Handling 136
11.4.1. Exceptions Raised During the Execution of Statements 136
11.4.2. Exceptions Raised During the Elaboration of Declarations 138
11.5. Exceptions Raised During Task Communication: Removed 139
11.6. Exceptions and Optimization: Removed 139
11.7. Suppressing Checks: Removed 139

Chapter 12. Generic Units: Removed 141

Chapter 13. Representation Clauses and Implementation-Dependent Features 143

13.1. Representation Clauses: Removed 143
13.2. Length Clauses: Removed 143
13.3. Enumeration Representation Clauses: Removed 144
13.4. Record Representation Clauses: Removed 144
13.5. Address Clauses: Removed 144
13.5.1. Interrupts: Removed 145
13.6. Change of Representation: Removed 145
13.7. The Package AVA 145
13.7.1. System-Dependent Named Numbers 146
13.7.2. Representation Attributes: Removed 146
13.7.3. Representation Attributes of Real Types: Removed 147
13.8. Machine Code Insertions: Removed 147
13.9. Interface to Other Languages: Removed 147
13.10. Unchecked Programming: Removed 147

Chapter 14. Input-Output 149

14.1. External Files and File Objects 149
14.2. Sequential and Direct Files 150
14.2.1. File Management 151
14.2.2. Sequential Input-Output 151
14.2.3. Specification of the Package Sequential_IO: Removed 151
14.2.4. Direct Input-Output: Removed 151
14.2.5. Specification of the Package Direct_IO: Removed 152
14.3. Text Input-Output 152
14.3.1. File Management 152
14.3.2. Default Input and Output Files 153
14.3.3. Specification of Line and Page Lengths: Omitted 154
14.3.4. Operations on Columns, Lines, and Pages 154
14.3.5. Get and Put Procedures 154
14.3.6. Input-Output of Characters and Strings 155
14.3.7. Input-Output for Integer Types: Removed 156
14.3.8. Input-Output for Real Types: Removed 156
14.3.9. Input-Output for Enumeration Types: Removed 156
14.3.10. Specification of the Package AVA_IO 157
14.4. Exceptions in Input-Output 158

14.5. Specification of the Package IO_Exceptions: Removed 159
14.6. Low Level Input-Output: Removed 159
14.7. Example of Input-Output (Rewritten Example) 159

Appendix A. Predefined Language Attributes 161

Appendix B. Predefined Language Pragmas: Removed 165

Appendix C. Predefined Language Environment 167

Appendix D. Glossary 171

Appendix E. Syntax Summary 177

Appendix F. Implementation-Dependent Characteristics 189

Appendix G. INDEX 191

Appendix H. Postscript : Submission of Comments 219

List of Figures

List of Tables