The Ada Programming Language

J. Michael Beaver

University of North Alabama

Author Note

All discussions of the Ada programming language within this document refer to the Ada 2012 specification unless otherwise noted. Discussions referring to previous versions of Ada—namely, Ada 83, Ada 95, and Ada 2005—are indicated accordingly. For the most part, the discussion is kept as general as possible.

Table of Contents

History 3

Design Goals 3

Syntax 4

Basic Data Types and Type-Checking 11

Fundamental Data Structures 11

Scoping 14

Names and Expressions 17

Statements and Control Structures 18

Subprograms and Parameter Passing 24

Packages and Generic Units 29

Program Structure 30

Exceptions and Exception Handling 33

Concurrency 35

Recursion 40

Input and Output 40

Unusual Features 43

Contributions to the Programming Language Landscape 43

Global Issues 43

Promise for the Future 43

Conclusion 43

Appendix A: Sample Program 44

Appendix B: Complete Syntax Description 47

Appendix C: Language-Defined Checks 67

Appendix D: TITLE OF APPENDIX XX

Appendix E: TITLE OF APPENDIX XX

Appendix F: TITLE OF APPENDIX XX

Appendix G: Example Generic Package XX

References 69

The Ada Programming Language

[Introduction]

**History**

The history of the Ada programming language is ultimately the story of the United States Department of Defense’s (DoD) attempt to develop a powerful language for embedded systems, especially for military applications. With the rise of software costs and the proliferation of non-standardized programming languages in DoD projects in the 1970s, there was a real need to develop a standardized language [2,7,12]. The original design process for Ada began in 1975 with the formation of the High-Order Language Working Group (HOLWG), which included representatives from all the military branches. In April 1975, the HOLWG produced a document known as Strawman that was distributed to relevant groups in the United States and Europe. The Strawman document was followed by revised versions: Woodenman in August 1975; Tinman in January 1976; Ironman in January 1977; and, Steelman in June 1978 [2,12]. The Steelman requirements document then prompted a series of language proposals from different groups.

Of the proposed languages, all of which were Pascal-based, the finalist was chosen in May 1979. The chosen language was proposed by a French team led by Jean Ichbiah working for Cii Honeywell/Bull. Interestingly, Ichbiah’s team was “the only foreign competitor” [2,12]. The name “Ada” itself was chosen in the same year in homage to Augusta Ada Byron, countess of Lovelace, the first programmer. From 1979 to 1982 the language specifications were drawn and revised; eventually a standard was recognized by the American National Standards Institute in 1983 [2,7,12]. The Ada standard was immediately mandated by the DoD, though this mandate was eventually lifted. To help enforce stabilization, the DoD also mandated that no supersets or subsets of Ada would be developed. This would combat the proliferation of language dialects, which was crippling the DoD’s development efforts in the 1970s [4,12].

The development of Ada was not an exercise in language science carried out by a specific vendor. Rather, development was a collaborative effort put forth by a community driven to engineer a language that emphasized reliability, maintenance, and the role of humans in programming [7]. Ada is a rather large programming language that attempts to provide something for everyone, in a sense. Quite simply, Ada was designed to solve practical problems in a real world; hence, Ada excels in its original domain of embedded systems programming. By the same token, Ada is much larger than its predecessor Pascal because it has to function as a practical language rather than as a training language [2,15]. Ada is not merely an extension of Pascal, though; it has introduced features not found in Pascal, such as the concept of the package. With respect to the concept of packages, Ada is related to SIMULA 67 and Modula-2; however, Ada was always Pascal-based [15].

**Design Goals**

Ada was originally developed with three concerns in mind: programs should be reliable and maintainable, programming is a human activity, and programs should be efficient. The 1995 revision introduced “greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas” [1]. The new 2012 revision of Ada provides even greater flexibility and more standardized packages [1]. Thus, Ada has become a significantly robust language in terms of predefined packages and support.

The emphasis on reliability and maintainability is fundamental to Ada programming. For this reason, the developers of Ada placed value on readability over writeability. This may be seen, for example, in Ada’s mandatory declarations of variables, which must be explicit and are invariant. This also allows the compiler to ensure that operations only occur on variables of compatible types [1]. What is more, the developers of Ada opted to remove error-prone and encoded notations in favor of a syntax that is decidedly English-like to ensure the readability of source code. In addition, the Ada programming language supports separate compilation of program units, which facilitates maintenance [1].

Ada’s developers recognized that programming is a human activity. It was out of respect for the human programmer that Ada’s developers opted “to keep to a relatively small number of underlying concepts integrated in a consistent and systematic way while continuing to avoid the pitfalls of excessive involution” [1]. The language was designed, in other words, to correspond to the expectations and intuitions of its users. To further facilitate ease-of-programming, Ada allows for programs to be assembled from independent components in the forms of packages, private types, and generic units. This modularization and separation also allows for programmers to update and modify their code without necessarily affecting already tested program units [1].

Efficiency—be it runtime, compile-time, cost-related, and so on—is always a concern for programmers and programming language developers. As such, it is imperative that languages be developed to be as efficient as possible. Since Ada is a compiled language, this meant designing the language in such a way that compilation was as efficient as possible. “Over-elaborate compilers,” according to the Ada 2012 reference manual, can “lead to the inefficient use of storage or [inefficient] execution time” [1]. Thus, the designers of Ada performed a systematic review of all the constructs within the language to ensure that the language was as efficient as possible. To this extent, any proposed language construct that failed to meet the specified efficiency criteria or had an unclear implementation was rejected by the designers [1].

**Syntax**

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

See Appendix B for a formal description of the syntax of the Ada programming language.

Ada was designed with preference given to readability over writeability. To achieve this, the designers of Ada introduced a number of reserved words (see Table 1). These reserved words have special meaning, and certain words may be attribute names [1]. The Ada 95 version introduced new reserved words that were not found in Ada 83: “abstract,” “aliased,” “protected,” “requeue,” “tagged,” and “until.” The Ada 2005 version also introduced new reserved words that are incompatible with the Ada 95 standard: “interface,” “overriding,” and “synchronized.” Finally, Ada 2012 has introduced the “some” reserved word, which is incompatible with previous versions of Ada [1]. Ada has over 70 reserved words, and each has a special meaning within the language’s syntax. These reserved words, of course, may not be redefined by the user; each is meant to enhance the readability of source programs.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 1: The reserved words in the Ada programming language as of the 2012 standard [1]. | | | |
| abort | abs | abstract | accept |
| access | aliased | all | and |
| array | at | begin | body |
| case | constant | declare | delay |
| delta | digits | do | else |
| elsif | end | entry | exception |
| exit | for | function | generic |
| goto | if | in | interface |
| is | limited | loop | mod |
| new | not | null | of |
| or | others | out | overriding |
| package | pragma | private | procedure |
| protected | raise | range | record |
| rem | renames | requeue | return |
| reverse | select | separate | some |
| subtype | synchronized | tagged | task |
| terminate | then | type | until |
| use | when | while | with |
| xor |  |  |  |

The Ada programming language, at least in its most recent revision, uses the character coding described by the ISO/IEC 10646:2011 Universal Multiple-Octet Coded Character Set standard. The coded representation for these characters is defined by the compiler implementer. In general, Ada supports uppercase, lowercase, titlecase, spacing, non-spacing, decimal characters, and others [1]. (See Table 2 for a listing of the categories of characters acceptable in Ada source code.) The lexical elements of Ada conform to the ISO/IEC 10646:2003 standard. This standard, according to the language designers, allows for support of what might be considered nontraditional characters, such as those found in the Korean language. Also, the ISO/IEC 10646:2003 standard provides case conversion tables, which is convenient since Ada is a case insensitive language, and it supports Unicode. All Ada implementations also must accept source code in UTF-8 encoding [1].

|  |  |
| --- | --- |
| Table 2: Categories of characters acceptable in Ada source programs. All descriptions refer to the General Category of ISO/IEC 10646:2011 [1]. | |
| **Category Title** | **Description** |
| letter\_uppercase | Any character whose General Category is defined to be “Letter, Uppercase.” |
| letter\_lowercase | Any character whose General Category is defined to be “Letter, Lowercase.” |
| letter\_titlecase | Any character whose General Category is defined to be “Letter, Titlecase.” |
| letter\_modifier | Any character whose General Category is defined to be “Letter, Modifier.” |
| letter\_other | Any character whose General Category is defined to be “Letter, Other.” |
| mark\_non\_spacing | Any character whose General Category is defined to be “Mark, Non-Spacing.” |
| mark\_spacing\_combining | Any character whose General Category is defined to be “Mark, Spacing Combining.” |
| number\_decimal | Any character whose General Category is defined to be “Number, Decimal.” |
| number\_letter | Any character whose General Category is defined to be “Number, Letter.” |
| punctuation\_connector | Any character whose General Category is defined to be “Punctuation, Connector.” |
| other\_format | Any character whose General Category is defined to be “Other, Format.” |
| separator\_space | Any character whose General Category is defined to be “Separator, Space.” The blank space is treated as a separator. |
| separator\_line | Any character whose General Category is defined to be “Separator, Line.” |
| separator\_paragraph | Any character whose General Category is defined to be “Separator, Paragraph.” |
| format\_effector | All character tabulation, line feed, line tabulation, form feed, carriage return, and next line characters. |
| other\_control | Any character whose General Category is defined to be “Other, Control,” and which is not defined to be a format\_effector. |
| other\_private\_use | Any character whose General Category is defined to be “Other, Private Use.” |
| other\_surrogate | Any character whose General Category is defined to be “Other, Surrogate.” |
| graphic\_character | Any character that is not in other\_control, other\_private\_use, other\_surrogate, and format\_effector. Notably: the quotation mark, the number sign, the ampersand, the apostrophe, the parentheses, the asterisk, the plus sign, the comma, the hyphen, the full stop, the colon, the semicolon, the less-than sign, the equals sign, the greater-than sign, the underline, the vertical line, the solidus (divide), the exclamation point, and the percent sign. |

Essentially every Ada program is constructed from one or more compilation units, which consist of sequences of lexical elements. Lexical elements are constructed from sequences of characters, and each element is “either a delimiter, an identifier, a reserved word,” a numeric literal, a character literal, a string literal, or a comment [1]. Comments have no bearing on the meaning of programs, but the meaning may be determined by the “particular sequences of lexical elements that form” the compilation units [1]. Each compilation unit may be divided into lines. The representation for the termination of lines is implementation defined, but a line must always be ended by a separator. Implementations of Ada must “support lines of at least 200 characters in length, not counting any characters used to signify the end of a line” [1]. This minimum is required by the language designers, but the maximum line length and the length of lexical elements may be defined by the implementer [1].

In the Ada programming language, identifiers are used as names for elements within the source program. Identifiers may consist of any uppercase, lowercase, or titlecase letters, modifiers, miscellaneous characters, punctuation, or numerals. Identifiers must begin, however, with an alphanumeric character, and they may neither contain consecutive punctuation characters nor end in such characters [1]. Identifiers in Ada must be unique and “are considered the same if they consist of the same sequence of characters after applying locale-independent simple case folding, as defined” by ISO/IEC 10646:2011 [1]. Identifiers, of course, may not be reserved words since reserved words have special meanings to the language and may not be redefined. Case mapping also means that identifiers may not be distinguished by arbitrary changes in uppercase and lowercase characters [1]. For example, the identifiers myString and MyString are considered to be the same identifier. Ada is case insensitive, so this consideration is logical.

Separators may be used to explicitly separate lexical elements within lines. The space character represents a separator, except when used within a comment, a string literal, or as a character literal. The character tabulation character may also serve as a separator, except within a comment. There is no limit to the number of separators between adjacent lexical elements. At least one separator is required, however, between adjacent identifiers, reserved words, and numeric literals [1].

Delimiters in Ada may be either simple or compound. Simple delimiters are the following characters: the ampersand, the apostrophe, the parentheses, the asterisk, the plus sign, the comma, the hyphen, the full stop, the solidus, the colon, the semicolon, the less-than sign, the equals sign, the greater-than sign, and the vertical bar. Compound delimiters are created by combining two adjacent simple delimiters. For example, the := (assignment) delimiter is a combination of a colon and an equals sign [1]. Note that, according to the 2012 reference manual, the square brackets and curly brackets, which are common delimiters in other languages, are not delimiters in the Ada language.

Comments may appear anywhere within an Ada source program. These comments are denoted by two leading adjacent hyphens, and comments extend to the end of a line. As with most programming languages, comments are not required for an Ada program to be legal. Also, comments have no effect on the semantics or meaning of the program. Comments’ sole purpose in Ada source programs is “the enlightenment of the human reader” [1]. Effective use of comments, of course, can considerably enhance the readability of source program.

Ada source programs may include numeric literals, which may be further divided into decimal and based literals. Certain numeric literals may be integer literals or real literals, which include a point. All decimal literals are presented in conventional base-ten notation. Decimal literals may include the digits 0 through 9, an underline, and the letter “E” to denote an exponent. The underline has no effect on the actual meaning of the decimal literal. The exponent character “E” can be either uppercase or lowercase because Ada is case insensitive. Exponents indicate the power of ten by which to multiply a decimal literal; exponents for decimal literals cannot be negative [1]. Whereas base-ten is assumed for decimal literals, a base must be explicitly given for based literals. Based literals also consist of the digits 0 through 9, an underline, and the letter E to denote an exponent, in addition to the extended digits A through F. Bases for these literals may be between two and sixteen. Based literals follow the conventional meaning of based notation. Bases and exponents should be written in decimal notation. An example of a based literal may be 16#FF#. Note that the number sign is used to separate the base (16) from the based numeral (FF); an exponent could be given after the second number sign. This particular example is equivalent to 255 in base-ten [1].

In addition to numeric literals, Ada supports character and string literals. Character literals are denoted in Ada by enclosing a character symbol between two apostrophe characters, like in the C++ programming language. A string literal, on the other hand, is a sequence of characters enclosed between two quotation marks, also like in the C++ programming language. Unlike C++, which uses the escape sequence \” to denote a single quotation mark *within* a string, the Ada language uses two adjacent quotation marks betweenbracketing quotation marks to denote a single quotation mark *within* a string literal [1]. For example, the string literal “””Hello,”” World!” would produce the string sequence “Hello,” World! Ada also supports null string literals, which are quotation marks with no characters between them [1].

Also like most high-level programming languages, Ada has support for language-defined compiler-directive pragmas. Pragmas may appear only in certain locations within a source program: “After a semicolon delimiter, but not within a formal\_part or discriminant\_part”; at any place (but not in place of) where a “declaration,” “item”, “statement,” “clause,” or “alternative” syntactic construct appears; at any place where a variant or exception handler syntactic construct appears; in place of a statement in a sequence of statements; in any compilation unit [1]. Pragmas are named by identifiers that follow the reserved word pragma, and pragma arguments follow the name of the pragma. Some pragmas may have specific identifiers with special meanings for that pragma [1]. In terms of static semantics, pragmas that are unrecognized by an implementation have “no effect on the semantics of the program”; implementations should provide warnings for unrecognized pragmas [1]. Pragmas may be executed if they appear “at the place of an executable construct”; execution entails evaluating each “pragma argument in an arbitrary order” [1]. Implementers may also provide implementation-specific pragmas as long as those pragmas do not adversely affect standard pragmas or otherwise legal programs [1].

Ada provides three particularly noteworthy pragmas. The pragma List takes on a single argument, either On or Off, and “specifies that listing of the compilation is to be continued or suspended until a List pragma with the opposite argument is given within the same compilation” [1]. The pragma Page essentially indicates a page break where the code following the pragma shall start on a new page if a listing is being generated. The pragma Optimize, perhaps the most important, takes one of three arguments: Time, Space, or Off. This pragma has effect “until the end of the immediately enclosing declarative region” or until the end of a compilation unit. This pragma notifies the compiler whether time optimization or space optimization is more important; also, priority optimization may be turned off [1]. Annex L of the 2012 reference manual provides a considerable list of language-defined pragmas.

**Basic Data Types and Type-Checking**

In an Ada program, named entities like constants and variables must be declared by declarations. The declaration associates an identifier (the name) with an entity, and declarations may be explicit or implicit. All declarations “contain a *definition* for a *view* of an entity,” which consists of the entity’s identifier and the entity’s view-specific characteristics. For each declaration, there is also a certain scope of the declaration; hence, declared identifiers are valid only within scope. Declarations come into effect at runtime by the process of elaboration [1].

Most declarations involve some type information. A type, in Ada, is described by a set of values and a set of operations. An object is a runtime entity of a given type and has the value of that type [1]. In Ada, types are grouped into categories of types, and there exist several language-defined types. Types are usually elementary or composite; the latter are composed of component values. The elementary types in Ada are the scalar types and access types. Scalar types are further divided into integer types, enumeration types, floating-point types, and fixed-point types. Access types are used to access objects and subprograms. The most notable composite types are the array and record, which form the fundamental data structures in Ada (see the following section) [1]. Types may also have constraints that limit the range of possible values for an object of that type. By extension, users may declare subtypes that combine an original type, constraints, and some subtype-specific attributes [1].

Types may be declared with type declarations, which also include the type’s first subtype. Note that composite types cannot be recursive (i.e., a component is of the given type itself). A type defined by a type declaration is a named type because it has a naming identifier. Type declarations may also include type definitions [1]. For example:

**type** Color **is range** 1 .. 72;

In this example, a type named Color is declared, and it is defined to be a **range** of 1 .. 72. In addition to type declarations, subtypes may also be declared from previously declared types [1]. Suppose we wish to create a subtype of the type Color:

**subtype** Rainbow **is** Color **range** Red .. Blue;

In this example, Rainbow is a subtype of Color, and it is constrained to the range Red .. Blue [1]. The properties of subtypes may be further defined with predicate aspects. Subtype predicates are essentially Boolean values that enable or disable properties of a subtype [1]. Ada also supports derivation of types. That is, a derived type may be defined to derive characteristics from a parent type. For example, all numeric types are derived types since they are implicitly derived from a root numeric type. Derivation is similar to declaring a normal type except that the reserved word **new** is included as part of the definition [1].

Objects are named entities created at runtime and containing a value of a given type. Objects are typically created and initialized by elaborating declarations. Objects may be either constant or variable, of course. The value of an object is read when an object is evaluated, and the value is updated when assignment is performed on the object [1]. Object declarations are used to declare standalone objects and possibly initial values for those objects. The initialization expression in an object expression must, of course, evaluate to that object’s specified type. If an initialization expression is not provided, then the object is implicitly initialized. Any object declaration without the reserved word **constant** is considered variable; naturally, including the reserved word **constant** makes the object constant [1]. For example:

MyVariable : Integer;

MyConstant : **constant** Integer := 100;

These lines declare two objects: MyVariable and MyConstant. The former is an object of integer type, and the latter is a constant with initial value of 100.

The fundamental data types in the Ada programming language are the scalar types. The scalar types are classified as either discrete or numeric. The former consists of integers and enumerations, each of which has an integer position number. Numeric types are the integer and real types. All scalar types are ordered, which means the relational operators are predefined [1]. Scalar types have ranges, which have lower and upper bounds inclusively constraining the legal values of those types. A scalar type’s base range is the finite range of all values that may be represented by objects of that type. All scalar types also have a set of language-defined attributes (see Appendix D) [1].

An object of enumeration type is declared with a set of ordered, distinct elements. These elements are ordered and have a position number within the set, starting at zero. Enumeration literals (i.e., the elements) may be overloaded by specifying the same identifier more than once in the set of elements [1]. Consider the classic example of days of the week:

**type** Day **is** (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

Here we are declaring a type Day that is an enumeration with seven enumeration literals. Each enumeration literal has an integer position number. Hence, Monday corresponds to zero, Tuesday corresponds to one, and so on. Interestingly, an enumeration type is a character type “if at least one of its enumeration literals is a” character literal [1]. Character is itself a predefined type representing the 256 code points of Latin1 in the ISO/IEC 10646:2011 standard. Character literals themselves are enclosed in single quotation marks [1]. The standard Ada implementation also includes a special predefined enumeration type known as the Boolean type. The Boolean type has two enumeration literals True and False that are ordered so False is less than True [1].

The integer type is the other discrete scalar type. An integer type is defined by an integer type definition, and it is defined to be either signed or modular. A signed integer is as per convention. Modular integers work with all modular arithmetic using specified positive moduli; that is, modular integers are unsigned with “wrap-around semantics” [1]. The set of possible values for an integer is the infinite set of mathematical integers, namely **Z**; of course, due to representation constraints, the actual representable values are limited to a predefined maximum and minimum. Modular integers are unsigned, so they are limited to a range of 0 to one less than the modulus [1].

In addition to integers, Ada also includes real types that approximate the real numbers. There are two types of real numbers in Ada: floating-point and fixed-point. The former approximates the real numbers “with relative bounds on errors,” whereas the latter uses absolute bounds [1]. The error bounds for floating-point types are given by specifying the relative precision by specifying the minimum number of significant decimal digits. This number of decimal digits is indicated by the reserved word **digits**, and the value must be positive. Similar to the integer type, the floating-point type is theoretically limited only to the infinite set of rational numbers, namely **Q**. Of course, the actual values represented are limited by the machine representation of those values [1]. In contrast, fixed-point types do not necessarily specify a number of digits but a delta to set the absolute error bound. This value is denoted by the reserved word **delta**. Fixed-point types represent the set of values comprised of “the integral multiples of a number called the *small* of the type,” which is, of course, practically limited by the machine representation [1].

In response to modern developments in object-oriented programming, Ada has introduced tagged types and type extensions. These concepts are based on inheritance and polymorphism via dispatching operations at runtime [1]. Tagged types are denoted by the reserved word **tagged**, and every type extension is also a tagged type. Type extensions themselves are types derived from tagged types. Objects of tagged types have runtime-associated tags identifying the tagged type [1]. Every tagged type has a set of primitive subprograms (like normal types) that constitute its dispatching operations. A dispatching operation is called by using a statically or dynamically determined controlling tag that determines which subprogram body is executed [1].

There are special tagged types called abstract types and interface types. Abstract types may serve as the ancestors to other types but may not be instantiated as objects, similar to abstract classes in the C++ language. Abstract subprograms are subprograms that have no body but may be overridden when inherited, also similar as in the C++ language. The reserved word **abstract** is used to denote an abstract type or subprogram, and only tagged types are allowed to be abstract [1]. Interface types are abstract types that provide for a limited kind of multiple inheritances. Tagged types may have multiple interface types as ancestors [1].

An access type is a special type used to indirectly access a designated object or subprogram. Hence, there are two types of access types: access-to-object and access-to-subprogram [1]. Access-to-object types have associated storage pools, and multiple access types can share the same storage pool. A storage pool itself is a place in memory that collects objects dynamically created by allocators. Access-to-object types are further subdivided. Pool-specific access types have access only to the elements in their associated storage pools, whereas general access types have access to any element of any storage pool [1].

Ada is a strongly-typed language, and type-checking occurs at compile-time. As a consequence of this, of course, the programmer must explicitly specify all data items used in a program so the usage may be checked. Since Ada is strongly typed, then, it is logically impossible for any two distinct types to overlap, and data values must match the specified type [2,10,15]. Ada does allow for a very limited kind of type conversion to ensure type conformance. If a type is to be converted, it will never be converted to an unrelated type. In other words, type conversion is only to another closely related type, usually a subtype [1,10]. The obvious downfall is that Ada’s strong typing restricts the language’s flexibility [5]. This objection is moot, however, since Ada was designed specifically for reliability.

**Fundamental Data Structures**

There are two data structures fundamental to the Ada programming language. The array and record are composite data objects, but they are also the basic data structures of Ada. Arrays in Ada are composed of components of the same subtype, and each component is named by an index value of a specified discrete type. The composite values of the components make up the value of the array [1]. An array is characterized by its dimensionality, its upper and lower bounds, the type and position of the indices, and the components’ subtype. The ordering of the indices is significant since the array is a linear representation. Ada supports both one-dimensional and multidimensional arrays (in row-major form, unless the Fortran column-major form is specified). As is common in most modern high-level languages, Ada has support for strings, which are one-dimensional arrays of characters [1]. Interestingly, arrays may be declared unconstrained or constrained. Arrays of the former type do not have to give an explicit range, whereas the latter must provide an explicit range. An unconstrained array uses the compound delimiter <> (“box”) to denote the lack of constraint [1]. Consider the following examples:

**type** Vector **is array**(Integer **range** <>) **of** Real;

**type** Table **is array**(1 .. 10) **of** Integer;

These are examples of type declarations, but they help illustrate the difference between unconstrained and constrained arrays. Note that the first example Vector uses the box delimiter <> to denote a lack of constraint. Notice that the second example Table gives an explicit range of 1 .. 10, which means the array will have 10 elements.

Ada has some built-in operations, in the form of attributes, for arrays. The attribute First “denotes the lower bound of the first index range,” and the attribute First(N) “denotes the lower bound of the N-th index range” [1]. The attributes Last and Last(N) behave the same as First and First(N), except they denote the upper bounds, respectively. The attribute Range is equivalent to the range First .. Last, and the attribute Range(N) is the same as the range First(N) .. Last(N). Finally, the attribute Length “denotes the number of values of the first index range (zero for a null range),” and Length(N) “denotes the number of values of the N-th index range (zero for a null range)” [1].

Perhaps the best known data structure in Ada is the record, which was first introduced in COBOL. A record object is a composite object composed of named components, and the value of a record object is the composite value of its components’ values. Records typically consist of a list of components, but a null record is possible by using the null statement in place of a list of components [1]. Consider the following basic example taken from the 2012 reference manual:

**type** Date **is**

**record**

Day : Integer **range** 1 .. 31;

Month : Month\_Name;

Year : Integer **range** 0 .. 4000;

**end record**;

This example excerpt creates a type Date which may be used in variable declarations. The Date object itself consists of three components: a Day field, a Month field, and a Year field. Notice that this record consists of two integer types and a presumed enumerated type. Also notices that the integers have their own constrained ranges of values. These fields may be access and updated by an object of the Date type after such an object has been declared.

Interestingly, records may include an optional variant part, which specifies an alternative list of components. Each variant specifies the components for the values of the record’s discriminant as covered by its discrete choice list. The choices must be static, but Ada does allow the use of an others section to serve as a catch-all. In the event of null records, the record’s variant, of course, will have no components [1]. The concept of variants is best shown with an example. Consider the following example:

**type** Device **is** (Printer, Disk, Drum);

**type** State **is** (Open, Closed);

**type** Peripheral(Unit : Device := Disk) **is**

**record**

Status : State;

**case** Unit **is**

**when** Printer =>

Line\_Count : Integer **range** 1 .. Page\_Size;

**when others** =>

Cylinder : Cylinder\_Index;

Track : Track\_Number;

**end case**;

**end record**;

In this example we have a record Peripheral that has a variant part. Notice that the variant part is merely a case structure and behaves as such. In this example we have an explicit alternative Printer, and we have a catch-all others alternative. Depending on the value of Unit, the record will behave differently. If the Printer alternative is taken, then we declare an integer Line\_Count; otherwise, we declare the variables Cylinder and Track if Unit is Disk or Drum.

Composite types, except arrays, may have discriminants, which parameterize the type of the composite type. Discriminants are components of objects and are of either a discrete type or of an access type. If a type is a derived type and has a declared discriminant, then the subtype of the parent is constrained. Also, if the parent is not tagged, then each discriminant of the derived type is used in the constraint of the parent subtype. By extension, then, the subtype of the constraining discriminant must be statically compatible “with the subtype of the corresponding parent discriminant” [1]. Thus, a discriminant can be used to constrain the values of a discriminated type (i.e., a composite type with a discriminant).

**Scoping**

The Ada programming language has specific rules for the scoping of declarations and the visibility of identifiers. Certain portions of Ada programs form declarative regions. These include block statements, loop statements, accept statements, exception handlers, and so on. Declarations immediately within a declarative region are said to be local to that declarative region; hence, entities are local to a declarative region if they are declared in that region. A declaration is said to be global to a declarative region if its declaration occurs within an encapsulating declarative region. Hence, any entities that are declared globally are global to the declarative region [1].

For each declaration, there is a a scope of the declaration within the program text; this scope applies also to the entities declared by the declaration. Declared entities may be referenced only where they have scope [1]. A declaration’s immediate scope is the part of the declarative region enclosing the declaration. Typically this immediate scope extends from the beginning of the declaration to the end of the declarative region. The immediate scope of the declaration, then, is also the immediate scope of the declared entity. An entity’s visible part is a portion of its declarative text visible from outside. The private part of an entity, on the other hand, contains all he declarative information that is not visible from outside. Only callable entities, other program units, and composite types may have visible and private parts [1].

Ada has visibility rules that govern which declarations are visible at different places within programs. These rules apply to both explicit and implicit declarations. A declaration is said to be directly visible when an identifier or operator symbol is sufficient to denote the name of the declaration. A declaration is visible, then, wherever it is directly visible and where some name can denote the declaration [1]. Direct visibility may be divided into immediate visibility and use-visibility. The former applies when the declaration is directly visible in its immediate scope. The latter applies only when the declaration is directly visible because of a use clause. A declaration can fall under both conditions [1]. A declaration may be hidden from visibility within parts of its scope. The declaration may be fully hidden, meaning it is not visible at all, or it may be hidden only from direct visibility [1]. Two or more declarations may be overloaded. These declarations are said to be overloaded if they have the same name and there exists a place where they are directly visible. Callable entities, such as subprograms, are also overloadable [1].

There is a special condition in Ada visibility known as overriding. This occurs when two type conformant, overloadable declarations have the same defining name. Typically this is not allowed immediately within the same declarative region unless one of the declarations is overridden by the other. Overriding is only possible, however, on “implicit declarations for predefined operators and inherited primitive subprograms” [1]. This means that any non-overridable declaration overrides an overridable declaration. An overridden declaration is also completely hidden from visibility within the scope of the overriding declaration [1]. The programmer has the ability to specify whether or not a declaration is overriding. This is done by including an overriding indicator with the declaration. An indicator of overriding indicates that overriding should occur, whereas an indicator of not overriding indicates that overriding should not occur [1].

The use clause is a special statement that brings declarations of specified packages and types into direct visibility. Each use clause has a certain scope within the program text, which may either be a declarative region or the entire body. In the former case, the scope extends from after the use clause until the end of the declarative region. In the latter case, the scope covers the entire body, of course. In the case of library units, if the scope of a use clause is in the private part, then the scope cannot be extended to the visible part of any public descendants of that unit [1]. For packages named by use clauses, any declarations immediately within the declarative region are potentially use-visible if the declaration is visible. For types, the declaration of each primitive operator is potentially use-visible if the declaration is within scope. By extension, then, any potentially use-visible declaration is use-visible unless there are naming conflicts [1].

Interestingly, Ada provides facilities for renaming declarations. These renaming declarations declare another name for entities such as objects, exceptions, packages, subprograms, and generic units. The renaming process is achieved by using the renames reserved word, and it determines the visibility of the renamed entity [1]. For example, to rename an object:

**declare**

B : MyObject **renames** A;

**begin**

B.MyContent := B.MyContent + 1;

**end**;

In this example the object A is renamed by the declaration for the object B. In the body, we increment the contents of B. A similar process is used to rename exceptions, packages, subprograms, and generic units, each using the renames reserved word [1].

Finally, the ability to overload presents the problem of multiple interpretations, which leads to unallowable ambiguity. The Ada language has a set of governing rules that determines which interpretation is legal; the rules are violated, then the interpretation is invalid. Understandably most type-checking rules also are overloading rules, and rules for matching of formal and actual parameters are overloading rules [1]. Interpretations are termed possible if they conform to syntax and visibility rules. Acceptable interpretations are possible interpretations that also conform to the overloading rules. There should be at least one acceptable interpretation, and if one exists, it is chosen [1].

**Names and Expressions**

All declared entities, subprograms, and objects in an Ada program have names. The evaluation of a name merely determines the entity denoted by that name [1]. Sometimes names are used to denote indexed components, which are components of arrays or entries in a family of entries. Elements in an array may be referenced by giving the name of the array as a prefix along with the index of the desired element in parentheses after the prefix. For example, Page(10) accesses the tenth element of the one-dimensional array Page. If the given index is not within the range of the array, a Constraint\_Error exception is raised [1]. As an alternative to accessing individual indexed components, slices of arrays may be constructed. A slice is a one-dimensional array constructed from a sequence of consecutive components of another one-dimensional array. A slice is expected to have the same type as the original array. Of course, if an attempt is made to slice an out-of-bound range, then a Constraint\_Error exception is raised [1]. For example, Page(1 .. 10) creates a slice of 10 elements from the original one-dimensional array Page.

Selected components denote components of entries, entry families, and subprograms. The selected component is called an expanded name if it denotes a package or an enclosing named construct according to visibility rules. Selected components that are not expanded names, however, resolve to: components; single entries, entry families, or subprograms; “a view of a subprogram whose first formal parameter is of a tagged type” [1]. Expanded names resolve to denote declarations occurring immediately within named declarative regions [1]. Selected components may be as simple as selected components from records. For example, MyRecord.MyComponent selects the MyComponent from the record MyRecord. Expanded names appear similar, but they must conform to the above rules. Suppose we have the unit MyUnit with the variable MyVar declared within its visible part. Then, we may denote this variable by using the expanded name MyUnit.MyVar in the text of our program.

An interesting feature of the Ada programming language is the concept of attributes. An attribute is “a characteristic of an entity that can be queried via an” attribute reference [1]. Attributes are designated by an apostrophe after an entity’s name followed by the attribute designator. Suppose we have an enumeration type Color, and suppose we want to access its attribute First. We may do this by writing Color’First, which denotes the attribute First. An attribute reference denotes a value, object, subprogram, or other entity. If an attribute designator includes an expression, it must be static and of integer type [1]. The standard Ada language implementation includes language-defined attributes (see Appendix D); however, implementers are allowed to create implementation-defined attributes. These implementation-defined attributes must differ from the language-defined attributes [1].

Like most programming languages, Ada allows for the use of literals, which represent values literally. A literal may either be numeric, character, string, or null. Numeric literals, of course, represent the name of the literal with its actual numeric value. Character literals represent one character and are wrapped in single quotation marks. String literals are merely an array of characters wrapped in double quotation marks. Null literals are null, meaning they do not have a numeric, character, or string representation [1]. Consider the literals ‘A’ and “ABC”. The character literal ‘A’ denotes only the character A, whereas the string literal “ABC” denotes the character array ABC.

Ada also has facilities for aggrandizing component values. In the Ada programming language, an aggregate combines component values into a record type, a record extension, or an array type. The evaluation of an aggregate necessitates the creation of an anonymous object to hold the obtained values from the components of the ancestor object [1]. With record aggregates, values for each component are specified either by named or positional association. This is best seen with an example:

(4, July, 1776)

(Month => July, Day => 4, Year => 1776)

In this example we have two record aggregates. The first aggregate has a component list that uses positional association, whereas the second aggregate uses named association [1]. Record extension aggregates are very similar to normal record aggregates, but they may be used to extend a record by including components not determined by the ancestor record. In this sense, the record is being extended to include new components [1]. In array aggregates, values are specified for each component of an array either by position or by index. In the former sense, the components are given in increasing-index order, whereas indices may be used in the latter sense. Array aggregates may be either one-dimensional or multi-dimensional [1]. For example:

(7, 1, 5, 3) *-- one-dimensional*

((1.1, 1.2, 1.3), (2.1, 2.2, 2.3)) *-- multi-dimensional*

The first example creates a one-dimensional array aggregate, and the second example creates a multi-dimensional array aggregate. Both of these examples use positional association.

In the Ada programming language, expressions are formulae that define the computation or the retrieval of some value. Syntactically, expressions are composed of choice expressions, choice relations, normal relations, simple expressions, terms, factors, primaries, conditional expressions, or quantified expressions [1]. (See Appendix B for the complete syntax description of an expression.) As is typical in high-level languages, names may be used in Ada expressions, and they must resolve to denote objects or values. Also, if an expression has a type, then “it specifies the computation or retrieval of a value of that type” [1]. Expressions may be as simple as arithmetical expressions like A + 4, or they may be as complex as parenthesized expressions like (A **and** B) **or** (**not** C), for example.

Most expressions contain operators in addition to primaries, or names of objects or values. In Ada, there are six categories of operators, in order of increasing precedence: logical operators; relational operators; binary adding operators; unary adding operators; multiply and division operators; and, exponentiation, absolute value, and **not**. Operators of the same precedence are evaluated left to right in order, and parentheses may be used to impose associations [1]. Each of the operators in the basic six categories is predefined in the Ada language, but users may redefine these operators by overloading. Also, operators may be used in traditional infix notation, or functional notation may be used [1]. For example, the expression A + B with the operator “+” may be written functionally as “+”(A, B).

An expression containing a logical operator resolves to some Boolean type. Ada provides the **and**, **or**, and **xor** logical operators already predefined, and these operators perform as per convention. These operators work on a bit-by-bit basis for the given operands. A value of zero represents false, and a value of one represents true. If the operands are arrays, then the operations are performed on a component-by-component basis [1]. In addition to the basic logical operators, Ada also provides the short-circuit control forms **and then** and **or else**. These forms behave the same as the predefined **and** and **or**, respectively, except that the left operand is always evaluated first. The right operand is evaluated only if necessary [1].

Ada has predefined relational operators and operators for testing membership. The predefined relational operators are equality =, inequality /=, less than <, less than or equal <=, greater than >, and greater than or equal >=. The operators **in** and **not in** are used to test whether a value belongs to a given subtype or range. The results of the relational operators and of membership test are of type Boolean [1]. When working with composite types, the equality operators work on a component-by-component basis. If there are no components, then the result is true. If there are unmatched components, then the result is false. Otherwise, the operators work with the matching components of the operands. The order in which the components are test is unspecified [1].

Ada has predefined binary operators. Binary addition and subtraction are defined in the conventional way for all numeric types. There is also the concatenation operator & that is predefined for all one-dimensional, non-limited array types, such as string literals. Concatenation behaves as expected: the contents of the right operand are appended to the contents of the left operand in a one-dimensional array whose length is the sum of the operands’ lengths [1]. For example, the expression “Fred” & “Wilma” concatenates the string literals “Fred” and “Wilma” to create the string object “FredWilma”. Ada also has predefined unary operators + (identity) and – (negation) in the conventional way for numeric types [1].

Ada has operators predefined for integer multiplication, division, modulus (**mod**), and remainder (**rem**). Signed integer multiplication behaves as per convention. Sign integer division and remainder have the relation A = (A / B) \* B + (A **rem** B), where (A **rem** B) has the sign of A and a magnitude less than that of B. The **mod** operator for signed integers is similar to the **rem** operator, except that the result is either zero or has the sign of the right operand [1]. Multiplication and division, but not remainder and modulus, are also defined for floating-point types. These operations have their conventional meanings, and accuracy is determined by the precision of the result’s type [1]. Of course, any attempts to divide by zero or use **mod** or **rem** with right operand of zero results in a Constraint\_Error exception [1].

There are three highest precedence operators in Ada expressions. The unary operator **abs** (absolute value) is predefined for numeric types [1]. The logical negation operator **not** is defined for all Boolean types, including arrays of Boolean type. In the case of logical negation of Boolean arrays, the result is an array whose components are the logical negation of the corresponding components of the operand [1]. Finally, the exponentiation operator \*\* is defined for all integer types and floating-point types. The exponent may be positive, zero, or negative, where the latter is defined as the reciprocal. In the case of negative exponents, a Constraint\_Error exception may be raised “if the intermediate result of the repeated multiplications is outside the safe range of the type, even though the final result (after taking the reciprocal) would not be” [1].

In addition to arithmetical expressions, Ada also has facilities for evaluating conditional expressions, notably if-expressions case-expressions. Upon evaluation, conditional expressions select one expression from a collection of enclosed dependent expressions. For if-expressions, the selection is made depending on some condition, whereas case-expressions make selections from alternatives based on some selecting expression. Ultimately, the condition of a conditional expression must be of any Boolean type [1]. Evaluation of if-expressions begins with the first if and continues through elsif alternatives until a final else, if it exists, is reached. If any condition is met, then the branch to the dependent expressions is taken immediately. For case-expressions, the selecting condition of the case is evaluated, and then the appropriate alternative is selected based on the evaluated value [1].

Type conversions in Ada are allowed only between closely related types, and conversion is always to a subtype of the original type. This is not implicit type conversion to force type-matching but explicit type conversion as specified by the programmer. Type conversions are either view conversions, as with the conversion of objects, or value conversions, as with numeric values. View conversions expect an object name as the argument for the conversion, whereas value conversions expect any expression as the argument [1]. When objects are converted by view conversions, the result is either a variable of the target type or a constant of a target type. The former can happen only if the operand is a variable. If an error occurs during conversion of an object, Constraint\_Error or Program\_Error may be raised. For value conversions, if the evaluated expression cannot be converted appropriately, then a Constraint\_Error exception is raised [1]. Consider the numeric type conversion Integer(1.6). In this example, the operand of the conversion is 1.6, and the conversion will be a value conversion. The result will be, which evaluates to 2. The fractional part is, of course, truncated [1].

Similar to the C++ language, Ada has an allocator expression that uses the **new** reserved word. Evaluation of an allocator expression “creates an object and yields an access value that designates the object” [1]. Evaluation of an allocator entails the evaluation of the accompanying expression, which occurs first. Upon evaluation of the accompanying expression, an object of the specified type is created, and then the value of the accompanying expression is converted and assigned to the object [1]. Consider the following example allocators:

**new** Cell’(0, **null**, **null**) *-- initialized*

**new** Cell *-- uninitialized*

In this example we have two allocator expressions creating two Cell objects. The difference between these allocators is that the first initializes the object according to the accompanying expression, whereas the second allocator leaves the object uninitialized [1]. Allocated objects really should be initialized before they are used in the program.

**Statements and Control Structures**

Statements define actions to be performed upon execution of the program. As with most modern high-level programming languages, Ada has two types of statements: simple and compound. Simple statements do not enclose other statements, whereas compound statements may enclose simple statements or other compound statements [1]. There are, of course, several types of statements, including simple null statements and compound loop statements. Statements may have statement identifiers, which must be distinct and appear in the same body as the statements. These statement identifiers are particularly useful for denoting labels, loop statements, or block statements. As might be expected, executing a null statement has no effect, and an implicit null statement follows any label that ends a sequence of statements [1]. Execution simply involves executing each successive statement until the sequence of statements is completed. This completion may happen when the sequence is terminated or when a transfer of control occurs. A transfer of control, in the Ada language, occurs when program execution is shifted to another location than what would have been normally expected. For example, a transfer of control could occur when an exception is encountered [1].

The assignment statement is a fundamental simple statement in the Ada language. Assignment is represented by the := delimiter, and when encountered, it “replaces the current value of a variable with the result of evaluating an expression” [1]. When an assignment statement is executed, the expression on the right-hand side is evaluated in an arbitrary order, and the calculated value is stored into the target variable on the left-hand side. The target may be any variable of any type that is within the appropriate scope. Since Ada is strongly-typed, the result of the expression on the right-hand side must be the same type as the variable on the left-hand side. According to the 2012 reference manual, “the value of the expression is converted to the subtype of the target,” which might cause an exception in some instances [1].

Example usage of the assignment statement may be similar to the following. Assume that U, N, and A are integer variables declared within scope:

N := 5;

A := 6;

U := N + A;

In this example, the integer values of five and six are stored into the variables N and A. Then, the expression N + A is evaluated. The integer result of 11 is then stored into the variable U, which is indeed an integer variable. If the result of the evaluated expression was not an integer, we might have expected an exception to be raised. Of course, this would be truly exceptional since the compiler should catch any type mismatches during compilation.

Ada allows for transfer of control with conditional logic via an if-statement construction. As with most high-level languages, at most one path within the if-statement block will be selected for execution based on the truth of a condition. Any if-statement block always starts with an if ... then construction, and an end if; signifies the termination of the block. There may be any number of elsif ... then (note the spelling) constructs within the block. There may be only one else construct within the block, and it is treated as an elsif True then construction. As expected, if a condition is true, then the corresponding sequence of statements is executed and the block exited afterward [1]. Consider the following example. Assume A, B, and C are previously declared integer variables within scope:

**if** A < B **then**

C := B;

**elsif** A > B **then**

C := A;

**else**

C := 0;

**end if**;

In this example, the variables A and B are compared in if and elsif constructions. If the first condition is true, the C is overwritten with the value stored in the variable B. However, if the first condition is false, we consider the second condition. If it is true, then C is overwritten with the value stored in the variable A. If both conditions are false, then C is overwritten to be zero.

In addition to if-statements, Ada also allows for case statements, which select from a number of alternative sequences of statements for execution based on some evaluated expression. This selecting expression must be of a discrete type, and the case options must be of the same type as the selecting expression. The case options must also distinct and static; an option of others may be provided alone in the list of options [1]. During execution, the selecting expression is evaluated first, and if a match is found in the list of options, the corresponding sequence of statements is executed. If an appropriate option does not match the evaluated selecting expression, an exception is raised. An others option may be provided to act as an analogue to the default option in the C++ programming language [1]. Consider the following example:

**case** Bin\_Number(Count) **is**

**when** 1 => Update\_Bin(1);

**when** 2 => Update\_Bin(2);

**when others** => **raise** Error;

**end case;**

In this example, the subprogram Bin\_Number returns a value based on Count that will determine which case is executed. The logic is straightforward and obvious; the others option serves as a catch-all to trigger an exception.

Ada allows for sequence of statements to be executed zero or more times repeatedly via a loop construction. There are three types of loops: a general loop with no iteration scheme; a loop with a while iteration scheme; and, a loop with a for iteration scheme. Loops may have unique identifiers that appear before the loop and are included as part of the end loop; statement [1]. Execution of a loop statement is considered complete when control is transferred out of the loop. For loops with the while iteration scheme, “the condition is evaluated before each execution of the” sequence of statements; the statements are executed only if the condition is true [1]. For loops with the for iteration scheme, the loop parameter specification is first elaborated, which creates the loop parameter and elaborates the discrete subtype definition. If a loop parameter has a null range, then the loop is complete; otherwise, the sequence of statements within the loop is executed once per iteration. The loop parameter is updated each time iterations complete. This update is usually performed in increasing order, unless the reverse reserved word is specified [1]. Consider the following example of a loop taken from the 2012 reference manual. Assume Next and Head are appropriately defined and are within scope:

Summation:

**while** Next /= Head **loop**

Sum := Sum + Next.Value;

Next := Next.Succ;

**end loop** Summation;

This loop has the label “Summation” and uses the while iteration scheme. This loop will continue executing while Next is not equal to (/=) Head. Notice also that the termination of the loop block includes the loop’s label.

Like with C++’s break statement, loops in Ada may be prematurely exited by using the exit statement. The termination may be absolute when the exit statement is encountered, or termination may be conditional if the exit statement has an accompanying condition [1]. The exit statement applies only to the loop immediately containing it. Upon execution, the condition, if present, is evaluated first. If the condition is true, control is transferred from the loop; otherwise, no transfer of control takes place [1]. Consider this augmented version of the previous example:

Summation:

**while** Next /= Head **loop**

Sum := Sum + Next.Value;

**exit when** Sum < 0;

Next := Next.Succ;

**end loop** Summation;

This new version of the “Summation” loop will now exit if Sum is zero. Otherwise, the exit statement will not be executed, and the loop will continue executing normally.

Block statements are perhaps as fundamental to the Ada language as the assignment statement, and the syntactic structure of the block statement is a clear holdover from Ada’s ancestor Pascal. A block statement essentially encloses a handled sequence of statements that may be preceded by an optional declarative part. If no explicit declarative part is included, then an empty declarative part is implicitly included [1]. Block statements are denoted by the word begin and are closed by the word end. A block statement may also have a distinct identifier, which must appear after the end if the identifier is used. Execution of a block statement begins with the elaboration of the declarative part, if it exists. Then the sequence of statements enclosed by the begin and end are executed [1]. Consider the following example. Assume that A and B are previously declared and are within scope:

Exchange:

**declare**

Temp : Integer;

**begin**

Temp := A;

A := B;

B := Temp;

**end** Exchange;

In this example, the block statement “Exchange” performs a simple swap using a temporary integer variable. Notice that the variable Temp has meaning only within this block statement.

Finally, the Ada language designers have given programmers considerable power by including an explicit goto statement. When executed, this statement will transfer control to a target designated by an accompanying label. This transfer automatically completes the execution of any compound statement that contains the goto statement “but does not enclose the target” [1]. This means that control may be transferred to “a statement of an enclosing” sequence of statements “but not the reverse”; it also means control cannot be transferred between alternatives, such as the options of a case statement [1]. Consider the following example taken from the 2012 reference manual. Assume the variable N is an appropriately declared integer and A is an appropriately declared array of some type within scope. Also, assume Exchange is an appropriately declared and defined subprogram:

<<Sort>>

**for** I in 1 .. N-1 **loop**

**if** A(I) > A(I + 1) **then**

Exchange(A(I), A(I + 1));

**goto** Sort;

**end if**;

**end loop**;

We have already seen how loops and if-statements work within the Ada language. Within this loop, within the if-statement, the goto statement will transfer control to the <<Sort>> label. Notice that labels for goto statements are enclosed in double angle-brackets.

**Subprograms and Parameter Passing**

Subprograms are an important construct in the Ada programming language. By definition, a subprogram is any program unit invoked by a subprogram call. In Ada, subprograms take one of two forms: procedures or functions. Procedures perform some action and do not return a value, whereas functions do return a value. Subprograms, at least in Ada, come in two parts: “a subprogram declaration defining its interface” and a subprogram body “defining its execution” [1]. A subprogram declaration declares a procedure or a function. Procedures are denoted by the indicator procedure, and functions are denoted by the indicator function and a return value. Any subprogram may have formal parameters that represent the actual parameters to be used in subprogram calls. Parameters have special modes: in (default), in out, and out. These modes convey “the direction of information transfer with the actual parameter” [1]. Special types of subprograms include abstract subprograms, null procedures, and expression functions. Subprograms may also be overridden by including the appropriate indicator [1]. Consider the following examples:

**procedure** Increment(X : **in out** Integer);

**function** Next\_Frame(K : Positive; Flag : **in** Boolean := True) **return** Frame;

Notice that all formal parameters have modes and types, and lists of parameters of different types are separated by semicolons. If no mode is specified, the mode in is implied. Also note that formal parameters of mode in may have default values [1]. Procedures have no return types, whereas functions must specify a return type.

Parameters, as mentioned, have special modes as specified in the subprogram declaration. Parameters may be passed either by copy or by reference. When a parameter is passed by copy, the formal parameter is separate from the actual parameter, which means that the transfer of information between the two parameters happens only before and after executing the subprogram. In other words, the contents of the actual parameter will not be affected by the subprogram. Parameters passed by reference, however, pass references to the actual parameter into the formal parameter. In this way, the actual parameter may be updated by the subprogram. Most elementary types are passed by copy; tasks, for example, are passed by reference [1]. The modes of parameters also affect how those parameters may be read or written. Parameters of mode in may be written to (i.e., passed into), but their contents may not be read once the subprogram terminates. The contents of parameters of mode out, on the other hand, may be read into an actual parameter before a subprogram terminates. Parameters of mode in out possess both qualities, acting almost like the pass-by-reference of the C++ language [1].

While a subprogram’s declaration specifies its interface, the subprogram body specifies a subprogram’s execution. Subprogram bodies are treated as declarations that either complete a previous declaration or serve as the initial declaration of a subprogram. A program may execute a subprogram body by invoking a subprogram call. Execution effectively has two phases: the declarative part is elaborated, and then the sequence of statements within the body is executed [1]. The interface profile in the subprogram body must conform to the interface form of the subprogram declaration. This means that both profiles must match in name, type, mode, and subtype [1]. Consider the following examples of a procedure body and a function body taken from the 2012 reference manual:

**procedure** Push(E : **in** Element\_Type; S : **in out** Stack) **is**

**begin**

**if** S.Index = S.Size **then**

**raise** Stack\_Overflow;

**else**

S.Index := S.Index + 1;

S.Space(S.Index) := E;

**end if**;

**end** Push;

**function** Dot\_Product(Left, Right : Vector) **return** Real **is**

Sum : Real := 0.0;

**begin**

Check(Left’First = Right’First **and** Left’Last = Right’Last);

**for** J **in** Left’Range **loop**

Sum := Sum + Left(J) \* Right(J);

**end loop**;

**return** Sum;

**end** Dot\_Product;

Notice that procedures do not have return types, and they do not need explicit return statements. Functions, on the other hand, must include a return type and at least one explicit return statement. Also, note that since the subprogram body is treated as a declaration, the local variable Sum in the function Dot\_Product is declared in the declarative portion of the body’s block statement.

If a subprogram is to be executed, it must first be invoked by a subprogram call either in the form of a procedure call statement or a function call. The subprogram call actually invokes the execution of the subprogram body, and it “specifies the association of the actual parameters, if any, with formal parameters of the subprogram” [1]. Each formal parameter for a subprogram can have at most one association with an actual parameter. Formal parameters without an association take on a default value [1]. Upon invocation, the name of the call is evaluation, and each parameter association is evaluated, including default parameters. The evaluations of these parameters occur in an arbitrary order. Once evaluations complete, the subprogram body is executed. “Any necessary assigning back of formal to actual parameters occurs” upon normal termination of the subprogram body [1].

Subprograms may be explicitly terminated by using a return statement. Procedures do not require a return statement, but a simple return statement may be used to perhaps prematurely terminate the procedure. Functions, on the other hand, must return a value, or a Program\_Error exception will be raised. Hence, the return statements in functions cannot be simple; they must include some value or expression to return [1]. Upon execution of a return statement, the accompanying expression is evaluated and converted to the result subtype. The converted result is then assigned to the return object. Note that this conversion could raise a Constraint\_Error exception if the value violates the subtype’s constraints [1]. Interestingly, return statements can be extended to include a sequence of statements to be executed during the return. This extended form behaves similarly to the simple return form, except that a sequence of statements is executed before control is transferred back to the calling entity [1]. Consider the following examples:

**return**; *-- In a procedure body*

**return** MyInteger; *-- In a function body*

**return** Node : Cell do *-- In a function body*

Node.Value := Result;

Node.Succ := Next\_node;

**end return**;

The first example is a simple return statement that can appear only in a procedure, not a function. It does not return a value; it merely transfers control to the caller. The second example also transfers control, but it also returns the value stored in MyInteger. The third example is an extended return statement. In this particular example we are updating values stored in a Node, but the extended form can be used to perform any sequence of statements. It seems to be most useful for updating the components of composite types. It is also worth noting that procedures can be specified to never return by specifying the No\_Return aspect in the interface. This prevents the procedure from ever returning normally, which may be useful for infinite loops or propagating exceptions. If the procedure does return normally, then a Program\_Error exception is raised [1].

Most powerful high-level programming languages incorporate the facility to overload operators, and Ada is no exception. Operators are functions whose designating identifiers are operator symbols; hence, like any other function, operators may be overloaded [1]. The use of unary or binary operators is, according to the 2012 reference manual, equivalent to a function call for that operator where the parameters are the operand(s) in order. Any function that specifies a unary or binary operator must, therefore, have one or two parameters, respectively, of mode in. Whereas normal functions may have parameters with default expressions, such parameters are disallowed for overloaded operator functions [1]. Conveniently, in the case of the equality and inequality operators, an explicit specification for the inequality operator is unnecessary. A declaration of the equality operator automatically generates an implicit inequality operator “that gives the complementary result” [1]. Consider the following examples taken from the 2012 reference manual:

**function** “+” (Left, Right : Vector) **return** Vector;

*-- Assume that A, B, and C are of type Vector*

A := B + C;

A := “+”(B, C);

In this example we overload the addition operator to work with two Vector entities and return the result as another Vector. Note that we may now use the operator + in infix notation or as the name of a function in a function call with the operands given as actual parameters.

Ada finally includes two special types of subprograms: null procedures (introduced in Ada 95) and expression functions (introduced in Ada 2012). A null procedure is exactly as its name implies; a null procedure has an empty body. Hence, the execution of a null procedure’s body has no effect [1]. An expression functions is convenient shorthand for a function whose body consists of a single return statement. The only effective difference between an expression function and a normal function is that the former consists merely of a return statement, which returns the evaluation of an expression [1].

**Packages**

The package is a fundamental program unit of the Ada programming language. Packages are program units that group logically related entities. Typically, packages contain the declaration of a type and primitive subprograms of the type that may be called from outside the package. The actual workings of the package are usually unknown to outside users [1]. In this way, the package is Ada’s answer to information hiding, similar in a way to C++’s concept of classes.

Packages are generally given in two parts: a package specification and a package body. All packages have package specifications, but not all packages need package bodies. The first list of declarative items in a package specification constitutes the visible part of the package. The reserved word **private** may be included to create a private part of the package; if no such part is designated, an implicit empty private part is added. Entities declared within the private part of a package are visible only within the declarative regions of the package and its children [1]. Whereas items declared in a package specification may be publicly visible, entities declared within a package body are visible only within the package body itself. Package bodies that are empty implicitly contain a null statement. Elaboration of a package body entails elaborating its declarative part first and then executing its sequence of statements [1]. The package specification, then, is analogous to the C++ class header, and the package body is analogous to the C++ class implementation.

Declaring a type as private within the visible part of a package confines its use to the package itself. This means that outside program units are not allowed to directly operate on the private data of the package [1]. Operations may be available for private types within a package either in the private part or in the body. These private operations are “available only inside the declarative region of the package” [1]. Private types in packages may also have type invariants. An invariant expression for a type may be of any Boolean type. Essentially, invariants are used in checks during evaluation; if an invariant expression evaluates to false, an Assertion\_Error exception is raised [1]. In addition to private types, packages may contain deferred constants. These are constants declared in the visible part of the package but whose value is given in the private part. Deferred constants are denoted by a **constant** declaration without an initialization expression. The actual value of the constant is given later in the private part of the package [1].

Every object that is instantiated from a package has three fundamental manipulations: initialization, finalization, and assignment. All objects are initialized explicitly by the programmer or by default. All objects are finalized before being destroyed. Assignment is used at minimum for assignment statements, explicit initialization, and parameter passing. The Ada language provides definitions for these operations, but users may implement their own versions when working with controlled types [1].

**Generic Units**

In the Ada programming language, there are generic program units that are either generic subprograms or generic packages. A generic unit in Ada, similar to a template in C++, is a template that may be parameterized and instantiated by other program units [1]. Generic units are declared by generic declarations, and instances of generic units are obtained through generic instantiation with the appropriate generic actual parameters for the generic formal parameters. An instance of a generic subprogram is a subprogram, and an instance of a generic package is a package [1]. Generic units are templates for other program units. Generic subprograms, for example, cannot be called directly but must be first instantiated. Once a non-generic version of a generic unit has been instantiated, the instance may be used as normal, but it cannot produce other non-generic instances [1].

Generic units come in two parts: a generic declaration and a generic body. The generic declaration declares a subprogram or package to be generic and contains any of the generic formal parameters. Generic formal parameters may be objects, types, subprograms, or packages [1]. The body of a generic unit serves as the template for instance bodies; the syntax for a generic body is the same as that for a non-generic body [1]. For a generic unit to be usable, however, it must be instantiated, by which an instance of the generic unit is declared. During instantiation, actual parameters are associated with the generic formal parameters of the generic unit. Any non-associated generic formal parameters are given a default value or expression. The instance of a generic unit is a copy of the template’s text with generic formal parameters replaced with the actual parameters [1].

Data may be passed to generic units either in the form of formal objects or as formal types. Formal objects are used to pass values or variables to a generic unit. These objects may be passed in either the **in** or **in out** modes. If no mode is specified, the **in** mode is implied [1]. Formal types can be used to pass subtypes of a certain category of types, which may be defined as all types in general. This includes private and derived types, scalar types, array types, access types, and interface types [1].

Even more powerful, Ada allows for the passing of formal subprograms and formal packages. Formal subprograms may be used to pass subprograms to a generic unit. This means that functions or procedures may be passed to a generic unit in a generic formal form to be actualized at instantiation [1]. Formal packages may be similarly used to pass packages to a generic unit. The formal package is replaced with the actual package upon instantiation. The actual package, then, is an instance of the template provided by the generic formal package [1]. See Appendix G for an example of a generic package.

**Program Structure**

In Ada, a program is “a set of partitions, each of which may execute in a separate address space, possibly on a separate computer” [1]. These partitions consist of library units. Programs may have one or more partitions, and multiple partitions may be executed concurrently [1]. One important aspect of Ada programs is the ability to compile units separately. Program units may either be packages, task units, protected units, generic units, or explicitly declared subprograms. Some of these program units may be compiled separately, or “they can appear physically nested within other program units,” such as a subprogram within a subprogram [1]. Program source code may be submitted to the compiler in one or more compilations, each of which is a succession of compilation units containing “the declaration, the body, or a renaming of a program unit” [1]. The language designers allow for the implementer to decide how to represent compilation units [1].

Library units are special types of separately compiled program units. Library units are always packages, subprograms, or generic units and may have library units as children or nested within them. Root library units together with their children constitute a subsystem [1]. Every library unit, except the Standard library, has a parent unit that is a library package or generic library package. In this hierarchy, all library units are children of the root library units, which are children of the predefined Standard library package [1]. Library units are designated by the prefix “library” or “generic library” if the unit is generic. The units’ declarations and bodies are also designated by the “library” prefix. Library units may have declared parent units; if none is specified, the parent is the Standard library. Any children of library units “occur immediately within the declarative region of the declaration of the library unit” [1]. Library units may also be declared private or made public. Use of the reserved word “private” indicates the unit is private, whereas its omission implies the unit is public [1].

Context clauses are used to specify which library items are needed with a compilation unit. This takes the form of “with” and “use” clauses. Depending on where the clause is located, “with” clauses may have scope of an entire declarative region of a library unit, or the scope may be only the body of the library unit. “With” clauses may also be private or limited and behave as might be expected [1]. “Use” clauses may be used to specify entities “within the declarative region” of library units or packages. In this way the user is able to explicitly bring an entity into full visibility [1].

Compilation units may be further divided into subunits. Subunits are very similar to child units. Subunits, however, support separate compilation of only bodies, not of declarations. Parent units must also explicitly indicate the location and existence of subunits. Declarations within a parent’s body are also visible in scope for any subunits of that parent [1]. Subunits identify their parent program units, and subunits may recursively identify other subunits as parents. Consider the following excerpt adapted from the Ada 2012 reference manual:

**package body** Parent **is**

Variable : String := “Hello World!”;

**procedure** Inner **is separate**;

**end** Parent;

**with** Ada.Text\_IO;

*-- subunit Inner with parent Parent*

**separate**(Parent)

**procedure** Inner **is**

**begin**

Ada.Text\_IO.Put\_Line(Variable);

**end** Inner;

In this excerpt, Inner is a subunit of the Parent package. Notice that within the package body of Parent, the procedure Inner is designated separate. This means that Inner is a separate subunit of Parent, and it must identify Parent as the parent unit. This is achieved with the separate command before the implementation of the Inner procedure.

At compile time, compilation units are compiled within certain contexts or environments. The contextual environment consists of “the outermost declarative region of the context” of any compilation unit [1]. Within an environment, declarative items appear in such a way that there are no forward semantic dependencies. That is, everything necessary for the items to be legally compiled is already established previously in the program source code or in included library units. The language designers have opted to let the implementer decide how exactly environments are created and how compilation units are added to environments [1].

Environments have special visibility rules. Within a parent unit at the start of a library item, only explicit library items of the environment are visible. Explicit root library items of the environment are the only declarations directly visible. If a limited “with” clause is used, only items that are implicitly declared in the limited view of a library package within the environment are visible. In these instances, only items implicitly declared within the limited view of a root library package are directly visible [1]. Library items within a library unit mentioned by a “with” clause may be made directly visible by including a “use” clause. In this case, any declarations within a visible declaration are made visible; this does not affect declarations that are not visible [1].

Execution of an Ada program entails execution of the set of partitions that constitutes that program. Each partition, in turn, contains an anonymous environment task, which contains declarative and body portions of the environment. Typically an environment task will include declarations of library items (arranged to avoid forward dependencies) and a body containing a sequence of statements. This sequence of statements usually includes “a call to the main subprogram, if the partition has one” [1]. In lieu of a call to the main subprogram, a null statement may be used. Execution of the partition, thus, entails execution its environment task. Execution of a partition ends when the environment task and the tasks contained therein complete normally or abnormally. Upon completion, the environment task “waits for the termination of all such tasks” and “finalizes any remaining objects of the partition” [1].

**Exceptions and Exception Handling**

Exceptions occur at runtime when errors or other exceptional situations arise. Raising an exception, therefore, diverts the program from its normal execution to draw attention to the fact that such a situation has arisen. Responding to these exceptions in some way is known as exception handling [1]. Ada provides facilities for both raising and handling exceptions. Exceptions may be raised explicitly by a raise statement or implicitly as a result of the failure of a language-defined check. Exceptions may be handled at runtime by a collection of exception handlers at the end of a block, or exceptions may be propagated to the containing block [1].

An exception may be explicitly declared, in which a name is given for the exception. Every exception declaration is distinct; hence, each exception has a unique identifier. The elaboration of an exception declaration has no effect [1]. The language designers have included several predefined exceptions in the Standard package. These predefined exceptions include: Constraint\_Error, Program\_Error, Storage\_Error, and Tasking\_Error. The Storage\_Error exception, for example, is raised when the execution of any construct has inefficient storage for execution [1]. The language designers have also included the ability for users to define their own exceptions. An identifier is given as with a variable, and the type is one of exception [1]. For example:

Fred : **exception**;

Wilma : **exception**

This code excerpt declares two exceptions: Fred and Wilma. The programmer has the ability to raise these exceptions within the appropriate scope.

Exceptions in Ada are always responded to by exception handlers. These exception handlers specify how the program is to respond to one or more exceptions at runtime. Programmers may specify which exceptions to handle, or others may be used as a catch-all to cover any unspecified exceptions. Exception handlers are also distinct in that no two exception handlers handle the same exception [1]. When an exception occurs, control is transferred to the appropriate exception handler. Then the associated sequence of statements for that handler is executed in response [1].

Exceptions also may be raised explicitly by the programmer. The raise statement combined with the name of an exception will raise the specified exception. These raise statements may be placed anywhere in the body of the program; interestingly, they may also be placed with no accompanying exception name in exception handlers as re-raises [1]. Once an exception is raised, program control is transferred to the appropriate exception handler. If no applicable exception is found within the current scope, the exception may be propagated to the containing scope, and so on. Normal execution is abandoned completely once an exception occurs. The executed sequence of statements within the exception handler effectively replaces the abandoned execution [1]. For an example of how to raise and handle exceptions, see the sample program in Appendix A.

Certain pragmas exist that allow the programmer to suppress or unsuppress language-defined checks. The Suppress pragma, when given a specified check, will omit that check, whereas the Unsuppress pragma “revokes the permission to omit checks” [1]. The Ada language standard features a set of language-defined checks (see Appendix C) that are designed to guarantee a certain level of runtime reliability and stability. These checks determine at runtime whether some condition is true; if the condition fails (is false), then an exception is raised [1]. Consider the following example:

**pragma** Suppress(Storage\_Check);

**pragma** Unsuppress(Index\_Check);

This excerpt uses the Suppress and Unsuppress pragmas. The Suppress pragma suppresses any checking for Storage\_Check-related errors; hence, exceptions might not be raised for those types of errors. The Unsuppress pragma disallows the suppression of Index\_Check-related errors; thus, exceptions might be raised for these errors.

**Concurrency**

Concurrency is fundamental to most Ada applications. Each Ada program during execution consists of one or more tasks, which represent separate, concurrent threads of control. These tasks may communicate synchronously or asynchronously. Tasks themselves, like other Ada units, are “defined by a corresponding task declaration” and task body, “which together define a program unit called a *task unit*” [1]. Similar to threads in other high-level programming languages, tasks in Ada proceed in stages. Initially a task is inactive, but it is later activated. Between activation and termination, a task is either blocked or ready to run. While a task is ready to run, it “competes for the available *execution resources* that it requires to run” [1].

Every task unit has a task declaration with a corresponding task body. Interestingly, tasks may be named or anonymous. Task declarations include task definitions, which include a list of task items. The task definition defines a task’s type and its first subtype, and the list of task items forms the visible part of the unit. A task may include a list of private list of task items, which is called the private part [1]. Consider the following example from the 2012 reference manual:

**task type** Server **is**

**entry** Next\_Work\_Item(WI : **in** Work\_Item);

**entry** Shut\_Down;

**end** Server;

This excerpt creates a task of type Server. There are two ways to enter this task: either through the Next\_Work\_Item point or through the Shut\_Down point.

Execution of a task of a certain type entails executing the corresponding task body. The initial stage of the execution is known of the activation of the task, which consists of the elaboration of the declarative part of the body. A task is said to have failed if an exception occurs during activation [1]. Tasks are represented by task objects (one task per object). Task objects may also activate other tasks; a task is blocked until all initiated tasks complete. If activation fails at any point, a Tasking\_Error exception is raised in the activating task. Furthermore, if an enclosing task terminates before a contained task, then the enclosed task is terminated [1]. All tasks, except the environment task, in Ada are dependent on one or more master tasks. This also means that tasks are dependent on the tasks that execute the master, and so on. Master tasks are blocked until all dependents have completed and terminated [1]. Consider the following example from the 2012 reference manual:

**declare**

**type** Global **is access** Server; *-- see above*

A, B : Server;

G : Global;

**begin**

*-- activation of A and B*

**declare**

**type** Local **is access** Server;

X : Global := **new** Server;

L : Local := **new** Server;

C : Server;

**begin**

*-- activation of C*

G := X;

. . .

**end**;*-- await termination of C and L*

. . .

**end**; *-- await termination of A, B, and G*

In this example we have a few Server tasks and two types (Global and Local) that are Servers. Upon the activation of the tasks A and B, we immediately activate other tasks X, L, and C. Then the tasks A, B, and G must wait for C and L to terminate. Once they have terminated, A, B, and G may resume execution or terminate.

A protected object is a special type of object that allows for access to shared date through calls on its visible protected operations, which may be protected subprograms or protected entries. There are also protected units, which are declared by a protected declaration and have a corresponding protected body [1]. Protected objects have special contents. Each protected object contains “the values of the components of the protected object, including (implicitly) an entry queue for each entry declared for the protected object” [1]. Each protected object also contains a representation of the execution states of the resources associated with the object. It is illegal to enter or call a subprogram of a protected object once it has terminated and finalized. If the error is detected, the Program\_Error exception is raised, or the task may be queued forever if the call proceeds normally [1].

Concurrent tasks in the Ada programming language communicate primarily via calls on entries and protected subprograms. The latter offers access to shared data objects, whereas the former allows for blocking of the caller until some condition is met, after which data may be shared directly or indirectly [1]. Protected subprograms are subprograms declared within protected definitions. Protected procedures provide exclusive read-write access to a protected object’s data, and protected functions provide concurrent read-only access to data. If a call is to an internal protected subprogram, then the call proceeds as if with a normal subprogram. However, if the call is to an external subprogram, then the subprogram’s body is executed as part of a protected action on the target protected object. The protected action is completed once the body of the subprogram terminates. A protected object may have only one protected action acting on it at a time, unless two or more protected actions are the result of a call to a protected function [1]. During a protected action, if an operation is potentially blocking, a bounded error results. If the bounded error is detected, then a Program\_Error exception is raised. If the error is not caught, then deadlock on a target object might result [1].

Operations on tasks and protected objects must be queued. This may be achieved via entry declarations with corresponding entry bodies or with accept statements. Actions to be performed upon the call of an entry are specified by the corresponding accept statement and by the corresponding entry body [1]. Execution of an accept statement entails identifying the appropriate entry to be accepted and then blocking execution until a corresponding caller is selected. Once this happens, execution of the accept statement’s own sequence of statements continues until exited. If an exception occurs during the execution of an accept statement’s sequence of statements, it is propagated to the corresponding entry call. The bodies of entries are executed once a corresponding caller has been selected, and the execution is similar to that of a normal subprogram [1]. This interaction between a calling task and an accepting task is known as a rendezvous. Once a rendezvous concludes, the tasks continue executing independently [1].

Entry calls may appear in different forms. Standalone entry calls are called simple entries and are unconditional calls on an entry of a task or a protected object. Upon execution, entries of tasks or protected objects are checked if they are available. If so, the call is selected immediately; otherwise, the entry call is added to an entry queue until selected or canceled [1]. Programmers can also specify that execution of a task be blocked for a specified period of time by using a delay statement (See Appendix A) [1].

Entries and accept statements may be paired with select statements. Select statements have four forms: a selective wait; select for timed entry; select for conditional entry; and, select for asynchronous transfer of control [1]. The selective accept variant allows for a task to wait and select from a number of alternatives; selection may depend on conditions associated with the alternatives. A timed entry call is an entry call that may be canceled if it is not selected before it expires. A conditional entry call is an entry call that may be canceled if it is not selected immediately; this might involve busy waiting with null statements. An asynchronous select statement allows for control to be transferred upon completion of an entry call or upon expiration of a delay [1]. Consider the following example of a selective accept taken from the 2012 reference manual:

**task body** Server **is**

Current\_Work\_Item : Work\_Item;

**begin**

**loop**

**select**

**accept** Next\_Work\_Item(WI : **in** Work\_Item) **do**

Current\_Work\_Item := WI;

**end**;

**or**

**accept** Shut\_Down;

**exit**; *-- Premature shutdown*

**or**

**terminate**; *-- Normal shutdown*

**end select**;

**end loop**;

**end** Server;

In this example we see a main accept statement and two alternatives. The main accept statement is executed if an appropriate entry call is selected. Similarly, the first alternative is selected if an entry call to Shut\_Down is accepted. Finally, if neither the main selection nor the first alternative is selected, the task is terminated normally.

Ideally tasks are allowed to terminate normally, but a task may be made abnormal by using an abort statement. Once a task is aborted, no other tasks may interact with it further. Abortion entails aborting the execution of the corresponding task body, unless it has already terminated [1]. Abortion of a task affects every construct within that task; all execution is aborted, unless the execution is abort-deferred. Certain operations are abort-deferred. These include protected actions, entry calls waiting to complete, dependent tasks that are terminating, among others. On the issue of dependence, the abortion of a master task necessarily aborts all its dependents [1].

Tasks also have the ability to communicate via shared variables. Two distinct, independent objects may be manipulated concurrently by two tasks without synchronization. However, tasks may be synchronized to allow for direct reading and writing of shared variables. Two tasks are synchronized when one task’s action signals an action of another task [1]. Of course, manipulating shared variables can result in erroneous data. To prevent this, actions on a shared variable must happen sequentially. This means that one action must signal another, the actions must be executed as part of the same task, or the actions must occur as protected actions on a protected object [1].

**Recursion**

Ada supports recursion with subprograms, packages, and types. All subprograms—that is, procedures and functions—may be recursively called. Calling a subprogram recursively is achieved simply by calling the subprogram like normal [1,5,7]. Ada also supports recursively declared packages and data types (and data structures); ideally recursion is avoided. To define a recursive package or type, however, the programmer must use pointers [7,13,14]. Recursion may also be disabled by using the Restrictions(No\_Recursion) pragma [1]. Consider the following example of a recursive type:

**type** Tree;

**type** TreeAccessType **is access** Tree;

**type** Tree **is** **record**

Left, Right: TreeAccessType;

Value: Integer;

**end record**;

This example recursively defines a record type Tree [13]. Notice that the forward declaration of the type Tree is followed by an access type (i.e., a pointer). This is the pointer that is used by the completed record declaration to define the Tree type. Hence, using the pointer TreeAccessType, the user is able to recursively define the Tree type.

**Input and Output**

Input-output functionality is provided by a set of language-defined packages. There are two generic packages Sequential\_IO and Direct\_IO that provide input-output support for files that contain elements of a specified type. The generic package Storage\_IO provides operations for reading from and writing to a memory buffer. There are also several packages such as Text\_IO that provide text-based input-output support. To accommodate exception-handling, there is also the IO\_Exceptions package that defines predefined input-output exceptions [1].  
 Ada supports input-output operations on external files. External files are any files “external to the program that can produce a value to be read or receive a value to be written” [1]. These files are identified by a name string and a form string. The latter indicates the system-dependent characteristics of a file, such as its access rights [1]. File input-output operations are not necessarily performed on external files themselves but on file objects that are associated with an external file. While a file object is associated with an external file, it is said to be open; otherwise, the file is said to be closed [1]. Files may be accessed either sequentially or directly; the appropriate operations are provided by Sequential\_IO and Direct\_IO, respectively. Naturally, files open for sequential access are termed sequential files, and files open for direct access are termed direct files. Files under sequential access are treated as sequences “of values that are transferred in order of their appearance” [1]. Files under direct access are treated as sets “of elements occupying consecutive positions in linear order; a value can be transferred to or from an element of the file at any selected position,” which is given by a positive index [1].

Ada also supports reading from and writing to memory buffers. The relevant operations are provided by the generic package Storage\_IO. It is worth noting that users can use this package to construct other input-output packages. For Storage\_IO, memory buffers have certain sizes “required to represent the content of an object” [1]. Reading from and writing to memory buffers is treated the same as working with direct access files; of course, the operations are on a memory buffer rather than a file object [1].

Human-readable text input-output is provided by the package Text\_IO. Characters are written sequentially and grouped into lines; sequences of lines are grouped into pages. Text input-output is similar to file input-output, but procedures Get and Put are used instead of Read and Write. There are overloaded versions of these procedures that can handle character, string, numeric, and enumeration types [1]. Text can be viewed as an array of characters. Each character in a line has a numbered index, starting from one. Similarly, lines of a page and pages of a file have numbered indices [1]. The programmer has the ability specify a maximum line length and a maximum page length. If a line exceeds the maximum length, then a new line is automatically created with the extra characters. A similar process happens for pages. As for manipulation of lines and pages, there are numerous functions and procedures provided in the Text\_IO package [1].

All possible exceptions arising from predefined input-output operations are defined within the IO\_Exceptions package. The possible exceptions are Status\_Error, Mode\_Error, Name\_Error, Use\_Error, Device\_Error, End\_Error, Data\_Error, and Layout\_Error. During execution, if more than one exception occurs, the exception that occurred earliest is the one propagated [1]. The Status\_Error exception arises when a program attempts to operate on a closed file or when an attempt is made to open an already open file. The Mode\_Error exception arises when operations are attempted on a file with an inappropriate file mode. For example, a file in mode In\_File cannot be written to, but it may be read from. If a program attempts to open or close a file that does not exist because of a bad filename, then the Name\_Error exception occurs. In the event that operations are attempted on a file with inappropriate characteristics, the Use\_Error exception results. On the system level, the Device\_Error is raised when there is a malfunction in the input-output device. During file reading, the End\_Error exception is propagated if an attempt is made to read past the end of the file. The Data\_Error exception is propagated if data read from a file does not match the required type or subtype. Finally, the Layout\_Error exception arises from errors in column, line, page, or string manipulation [1].

**Unusual Features**

To say a feature in the Ada programming language is unusual is perhaps a bit misleading because all the features of Ada are deliberately engineered. This being granted, there is one feature in Ada that is not necessarily common to other high-level programming languages. Programmers writing in Ada have the ability to write mixed-language programs that interface with C, COBOL, or FORTRAN [1]. Data or subprograms in another language may be accessed by an Ada program; conversely, Ada data and subprograms may be accessed in another language. Ada programmers can even specify that data and subprograms should conform to another language’s conventions [1]. There is no doubt that this is a powerful feature. To ensure reliability, Ada enforces its familiar restrictions on these interfaces. Thus, there is a kind of stability on either side of the interface, especially on the Ada side [1].

**Contributions to the Programming Language Landscape**

In the canon of programming languages development, Ada is a major milestone. Ada took the concepts introduced in Pascal, Modula-2, SIMULA, and others to a new level of reliability and maintenance. Ada also represents the first serious attempt by the United States Department of Defense to establish one standard language to be used for all its projects. (Ada later lost favor with the Department, unfortunately.) With its introduction, Ada brought some important contributions to modern programming languages [12].

With respects to reliability, exception handling is a critical aspect of software development. Exception handling was not a novel concept introduced by Ada, but Ada did help formalize and pioneer robust exception handling [2,9,12]. Indeed, according to Watt, “Ada was the first major language to include a secure form of exception handling” [14]. Ada’s implementation of exception handling, especially in its original form, was a major advancement over PL/I. For nearly a decade, until the 1990s, Ada was “the only widely used language that included this feature until the introduction of C++ and Java” [9].

Though object-oriented programming did not originate with Ada in the 1970s-1980s, the Ada programming language did mark a new development in objected-oriented programming. Ada was heavily derived from Pascal, which was not originally designed for object-oriented programming. In response, Ada introduced packages and mechanisms for derivation. With these features, Ada has been able to support inheritance, polymorphism, and dynamic binding, all important object-oriented concepts [12].

In addition to exception handling and object-oriented support, Ada also helped revolutionize concurrent programming with its use of tasks. This marked a boon for embedded systems programmers in the United States and abroad. Ada 83’s implementation of concurrency using the rendezvous was rather cumbersome, but Ada 95 solved this by introducing the task concept [12]. Tasks in Ada thus allowed for controlled access to shared data. Moreover, the data is encapsulated in a special structure that allows access only via rendezvous or subprogram call [12]. Thus, Ada, especially in its newer forms, has represented a revolution in concurrent programming, which is especially important for embedded systems programmers.

**Global Issues**

The Ada programming language, despite having been commissioned by the United States Department of Defense, is truly a global language. The original language proposal and specification were created by an international team led by a Frenchman, Jean Ichbiah [2,11]. Indeed, during the initial stages of development from 1975 to 1979, the effort to build Ada was international. The effort was conducted between American, British, and French governmental, industrial, and educational institutions. Moreover, the initial stages of development were punctuated with “literally thousands of comments from all over the world” [11,12].

Since its inception and standardization, Ada has taken a prominent role in global software development. For example, Ada is used heavily in avionics, air traffic control, and transportation development [6,12]. Ada is used literally all over the world, from Australia to Vietnam. With that in mind, Ada has adapted in newer versions to accommodate its international user base. Since not all users communicate in English, it makes sense that Ada should not be exclusively in English. Thus, as of at least Ada 95, the language has supported programming in languages other than English, which means official, standardized support for Unicode [1].

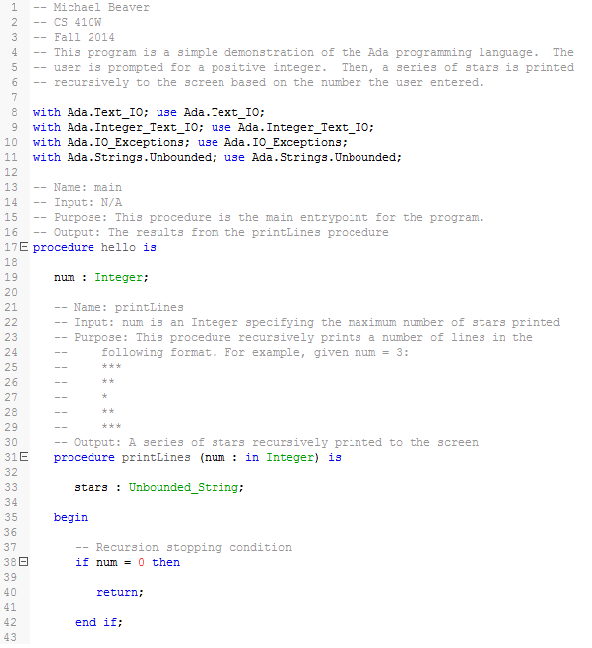
**Promise for the Future**

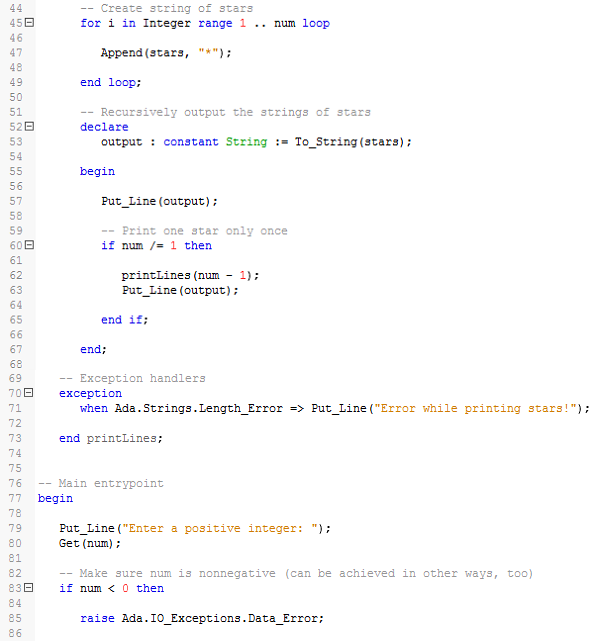
Like any good language, Ada has been in a constant state of development since its inception. Despite being standardized in various forms, the language continues to evolve, as evidenced by the many revisions to previous standards and the current standard [1]. With its massive global adoption, the future for Ada is bright. Ada’s future was not always so certain, however. In the 1980s and 1990s, Ada’s dominance was shaken by the introduction of newer object-oriented languages, such as C++. The defense at the time was that Ada supported object-oriented *design*, but it has since adapted to support object-oriented *programming* itself [12]. The current version, Ada 2012, has also expanded and has room to expand further yet; perhaps the most interesting feature is the ability to interface with other languages [1]. In terms of future developments, the Ada 2012 standard has fewer loose ends than the Ada 2005 standard. Interesting areas of future work include better support for nested packages and the introduction of shorthand assignment statements similar to C++’s shorthand assignment statements. As of now, no consensus has been reached on how to implement these features and others, but they are certainly on the horizon [3].

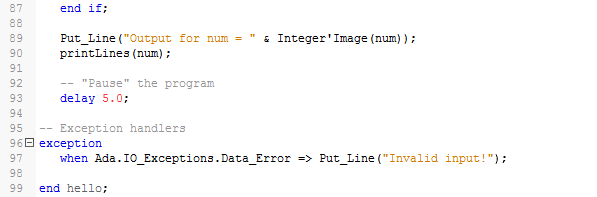
**Conclusion**

Appendix A

This appendix contains a sample program written in the Ada programming language. This program was written by the author using the GNAT Programming Studio integrated development environment.

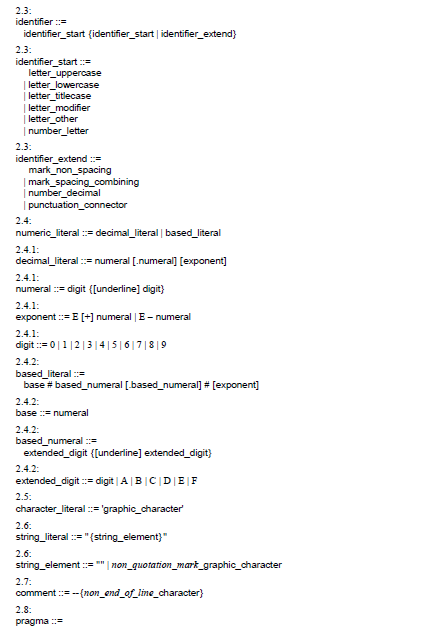


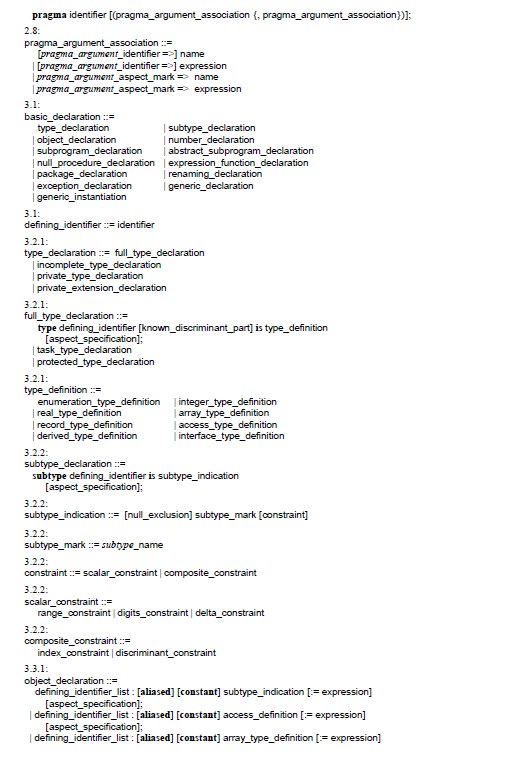


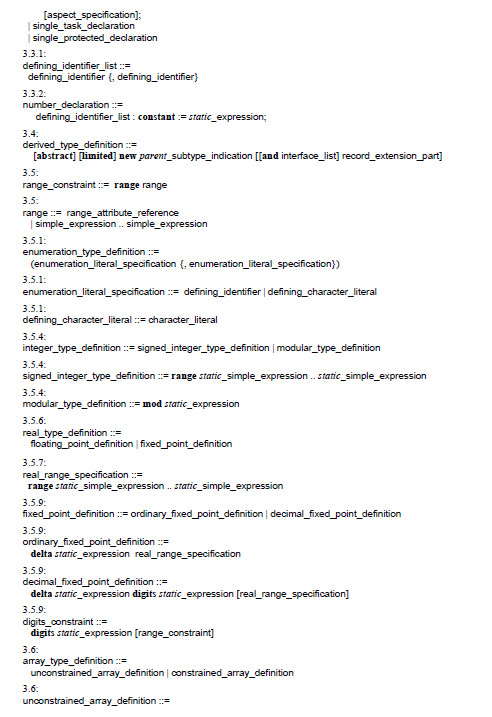


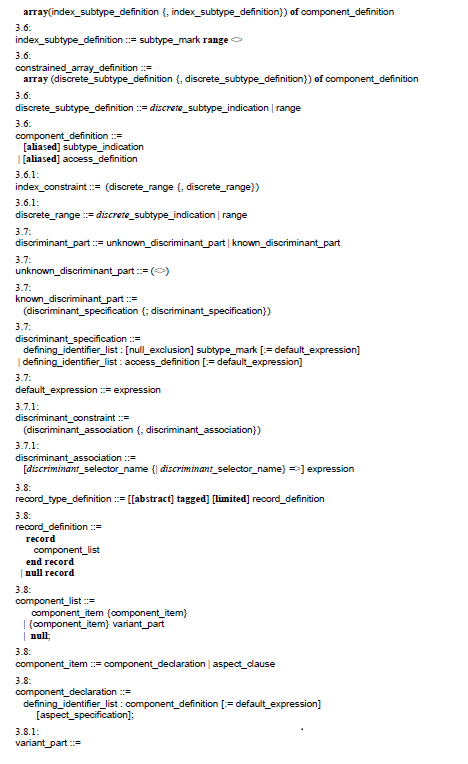
Appendix B

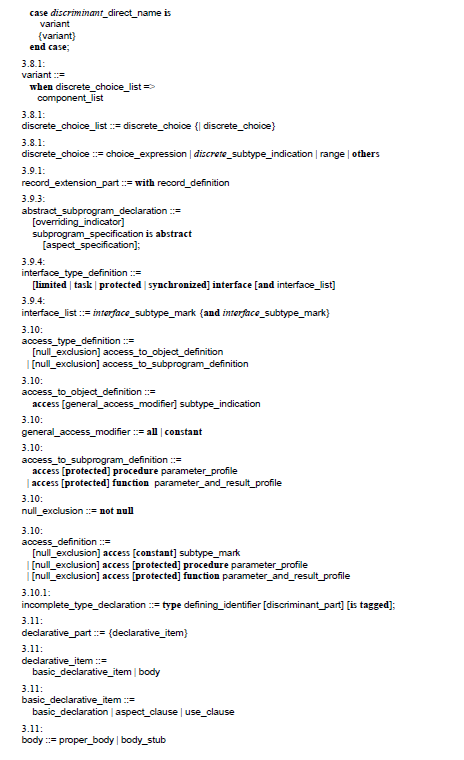
This appendix contains the complete Backus-Naur Form syntax description for the Ada programming language as given by the 2012 specification. The numbers before each entry are references to sections in the reference manual [1]. The contents of this appendix are reproduced exactly from the aforementioned manual.

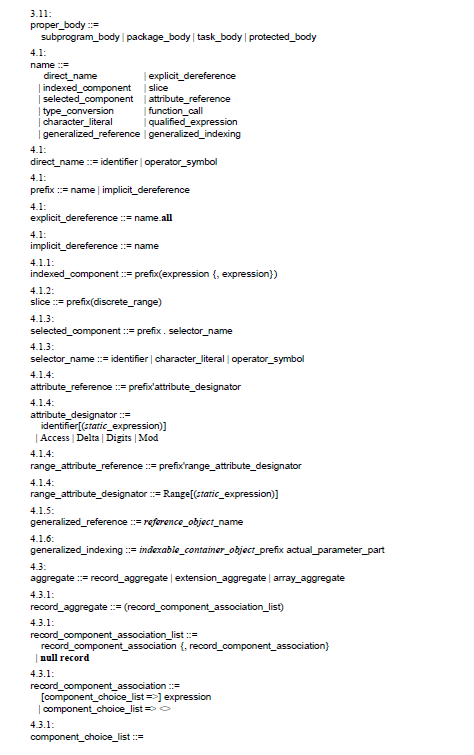


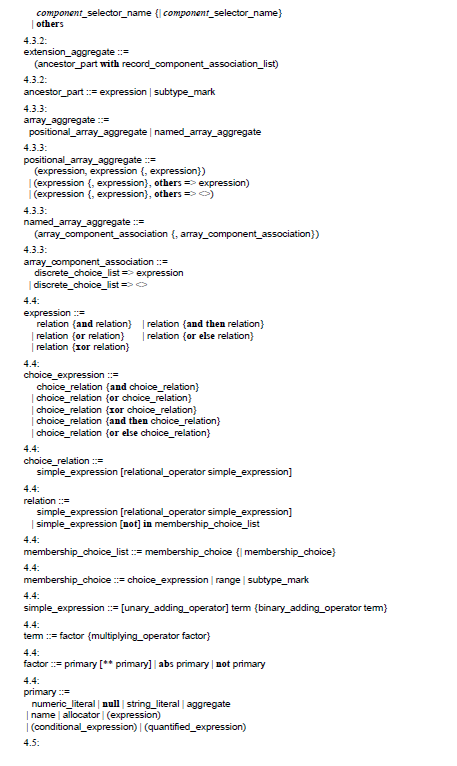


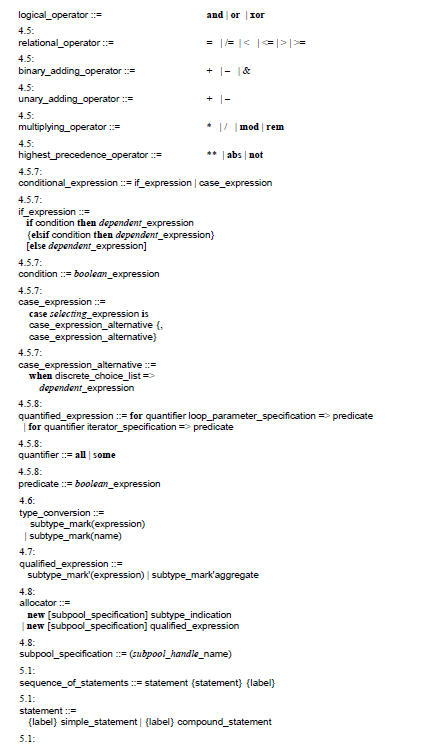


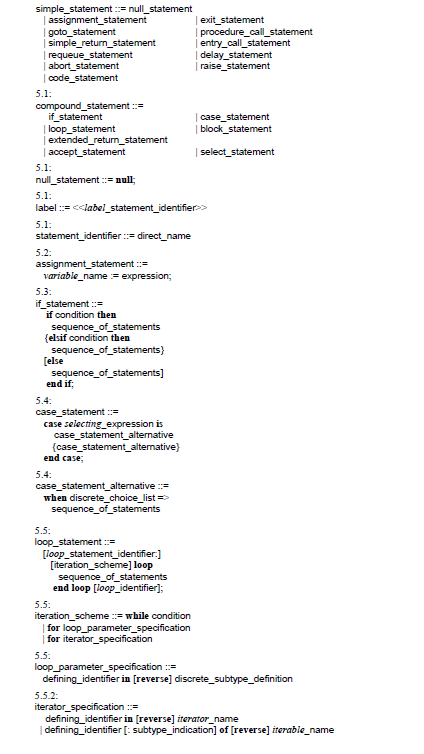


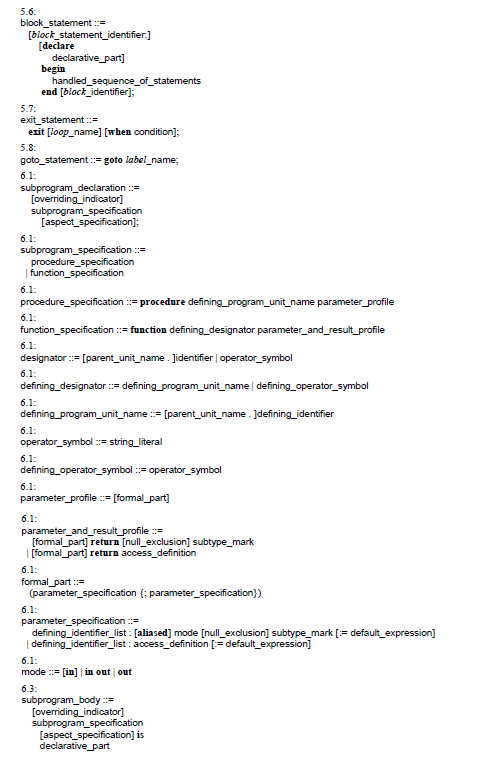


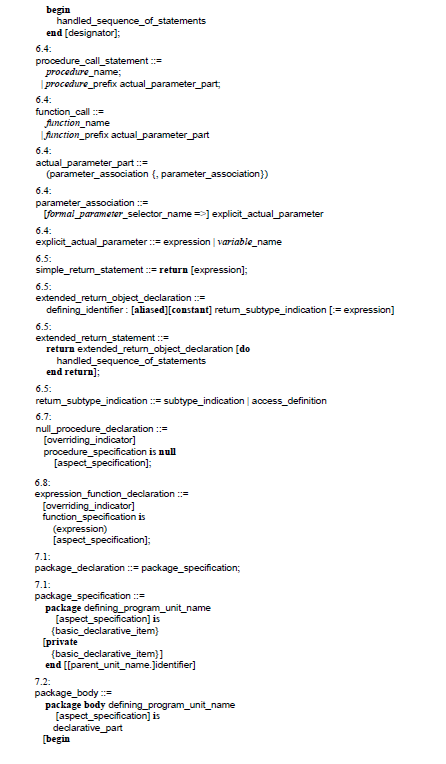


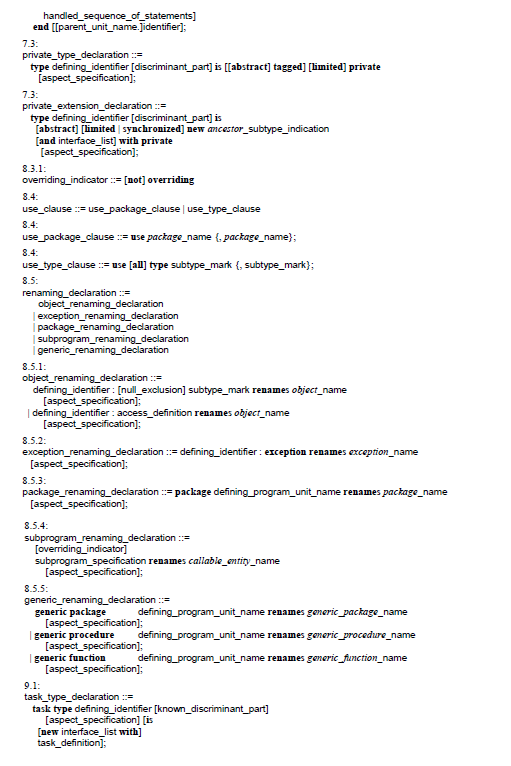


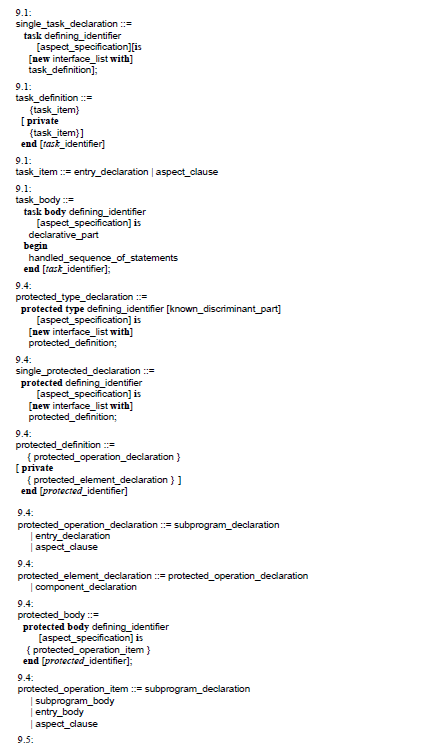


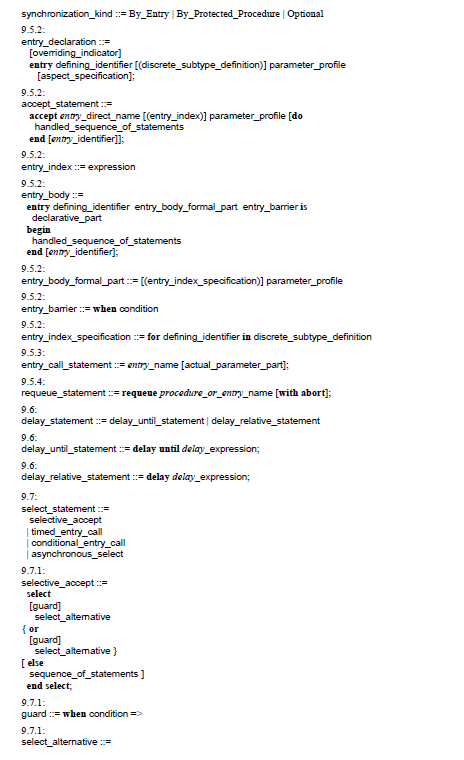


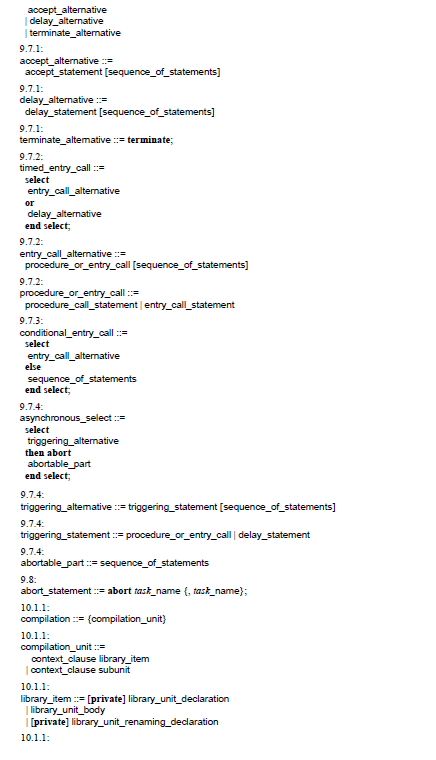


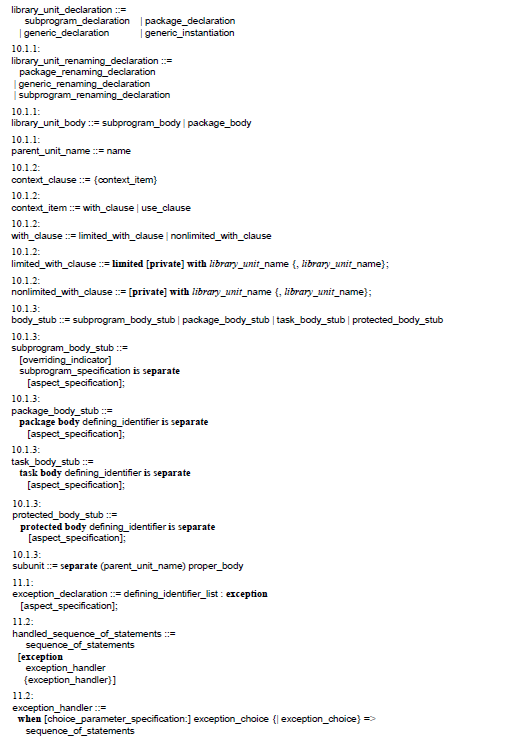


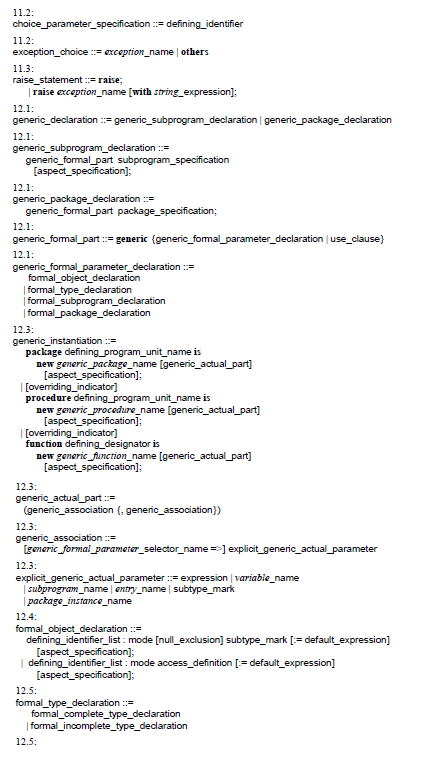


















Appendix C

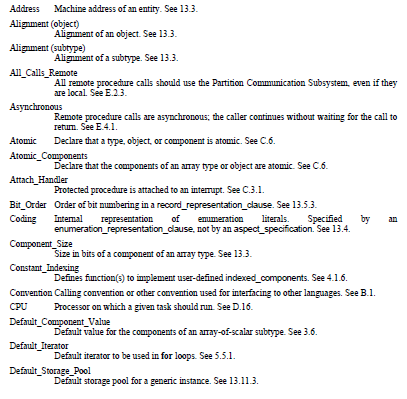
This appendix contains a table of all the possible language-defined checks that may be suppressed or unsuppressed by an implementation of Ada. The table contains the checks’ names, the exceptions raised by each check, and a description of each check as given by the 2012 reference manual.

|  |  |  |
| --- | --- | --- |
| Table C.1: The language-defined checks that may be suppressed or unsuppressed [1]. | | |
| **Check Name** | **Exception Raised** | **Description** |
| Access\_Check | Constraint\_Error | When evaluating a dereference (explicit or implicit), check that the value of the name is not null. When converting to a subtype that excludes null, check that the converted value is not null. |
| Discriminant\_Check | Constraint\_Error | Check that the discriminants of a composite value have the values imposed by a discriminant constant. Also, when accessing a record component, check that it exists for the current discriminant values. |
| Division\_Check | Constraint\_Error | Check that the second operand is not zero for the operations /, rem and mod. |
| Index\_Check | Constraint\_Error | Check that the bounds of an array value are equal to the corresponding bounds of an index constraint. Also, when accessing a component of an array object, check for each dimension that the given index value belongs to the range defined by the bounds of the array object. Also, when accessing a slice of an array object, check that the given discrete range is compatible with the range defined by the bounds of the array object. |
| Length\_Check | Constraint\_Error | Check that two arrays have matching components, in the case of array subtype conversions, and logical operators for arrays of Boolean components. |
| Overflow\_Check | Constraint\_Error | Check that a scalar value is within the base range of its type, in cases where the implementation chooses to raise an exception instead of returning the correct mathematical result. |
| Range\_Check | Constraint\_Error | Check that a scalar value satisfies a range constraint. Also, for the elaboration of a subtype indication, check that the constraint (if present) is compatible with the subtype denoted by the subtype mark. Also, for an aggregate, check that an index or discriminant value belongs to the corresponding subtype. Also, check that when the result of an operation yields an array, the value of each component belongs to the component subtype. |
| Tag\_Check | Constraint\_Error | Check that operand tags in a dispatching call are all equal. Check for the correct tag on tagged type conversions, for an assignment statement, and when returning a tagged limited object from a function. |
| Accessibility\_Check | Program\_Error | Check the accessibility level of an entity or view. |
| Allocation\_Check | Program\_Error | For an allocator, check that the master of any tasks to be created by the allocator is not yet completed or some dependents have not yet terminated, and that the finalization of the collection has not started. |
| Elaboration\_Check | Program\_Error | When a subprogram or protected entry is called, a task activation is accomplished, or a generic instantiation is elaborated, check that the body of the corresponding unit has already been elaborated. |
| Storage\_Check | Storage\_Error | Check that evaluation of an allocator does not require more space than is available for a storage pool. Check that the space available for a task or subprogram has not been exceeded. |
| All\_Checks | Any predefined exception | Represents the union of all checks. |

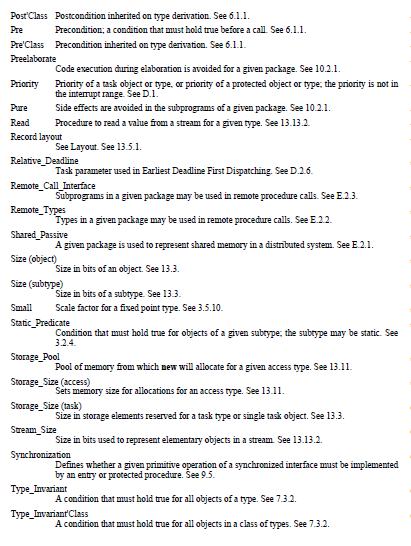
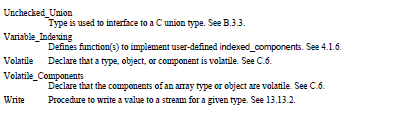
Appendix D

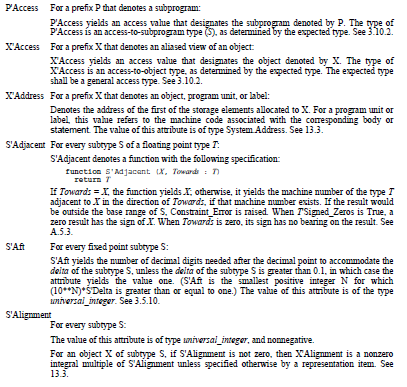
This appendix contains a listing of the language-defined aspects and attributes in Ada. There are cross-references to sections in the official 2012 reference manual [1].

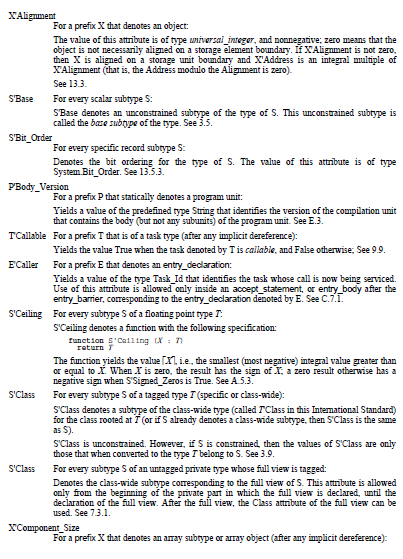
**Language-Defined Aspects**

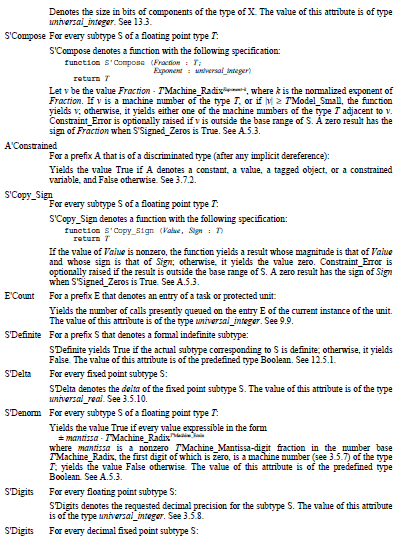


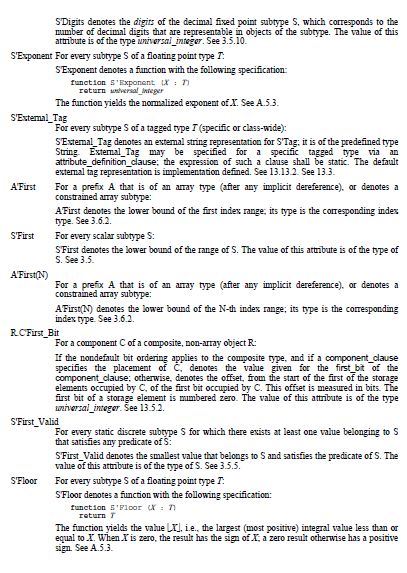


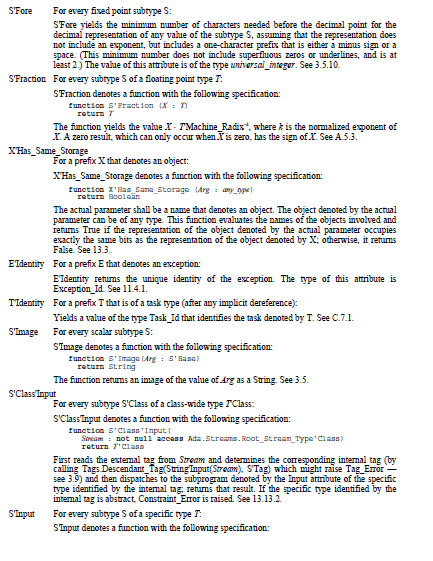


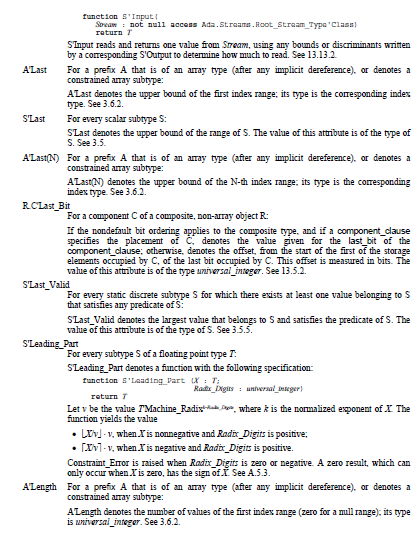
**Language-Defined Attributes**

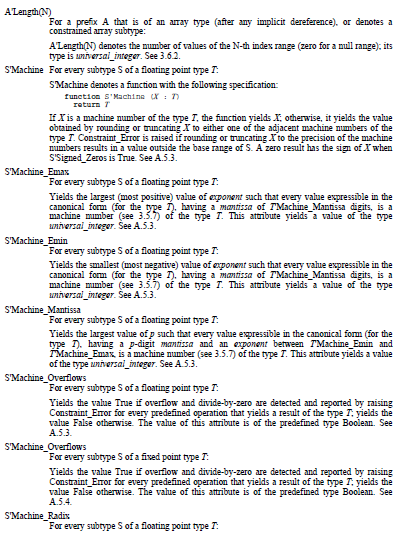


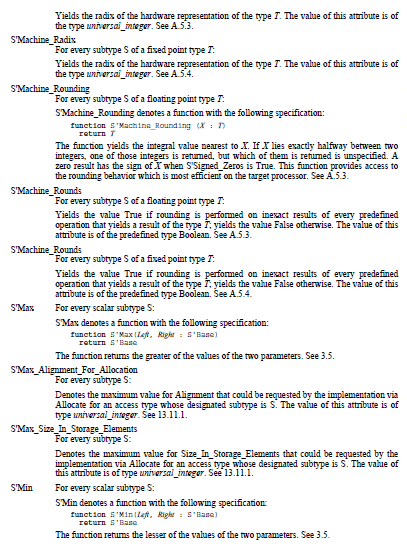


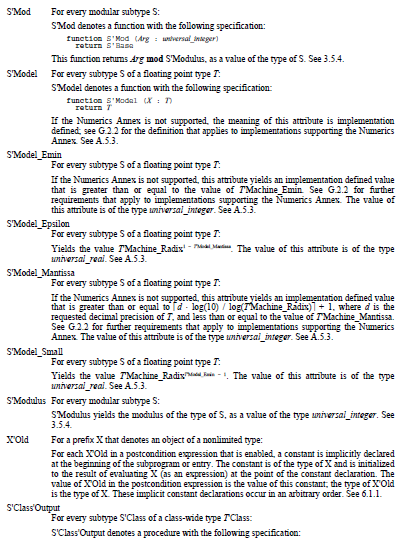


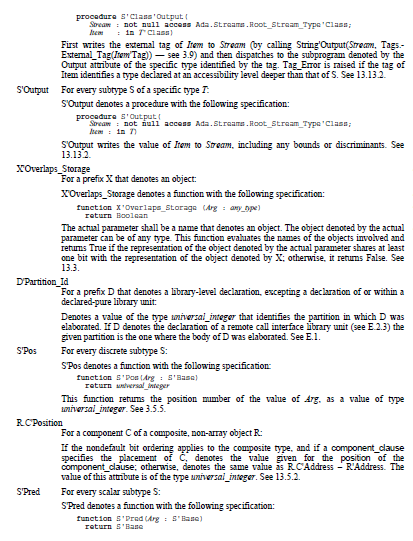


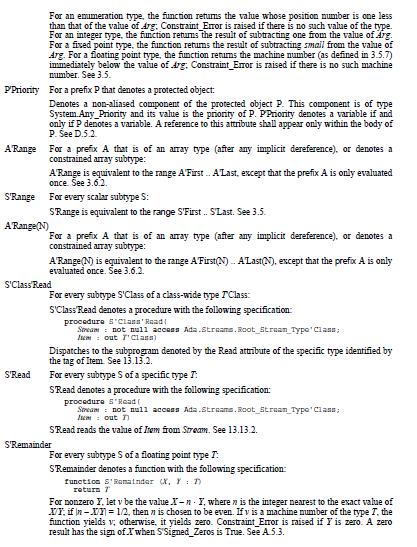


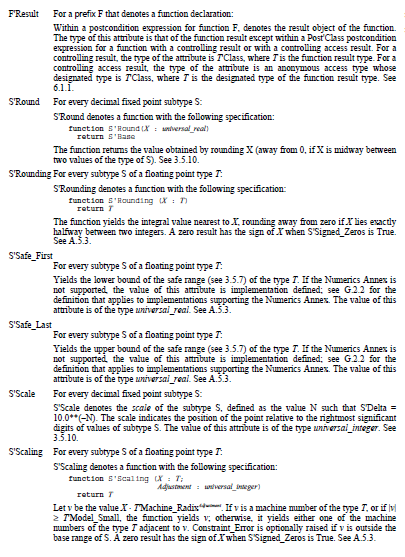


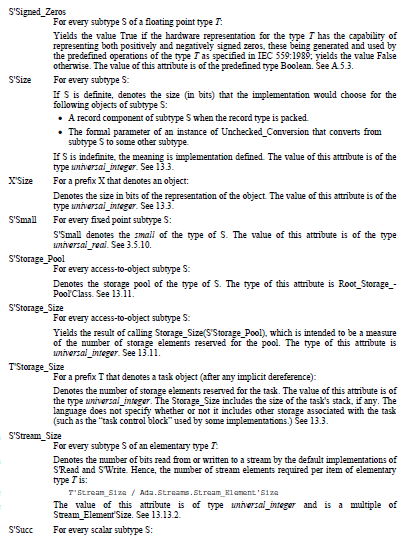


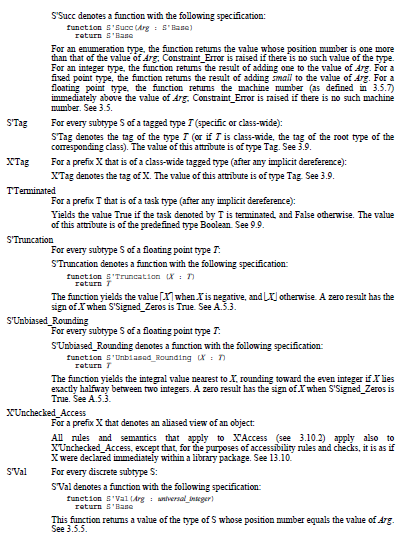


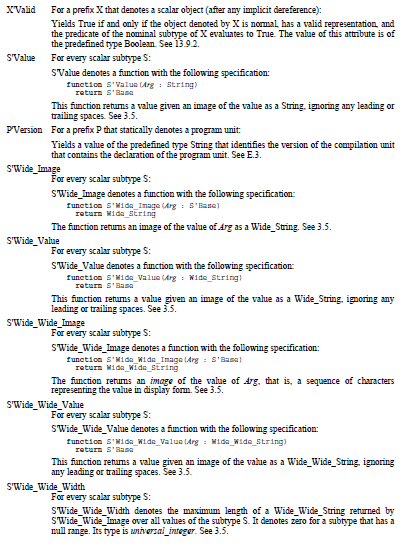


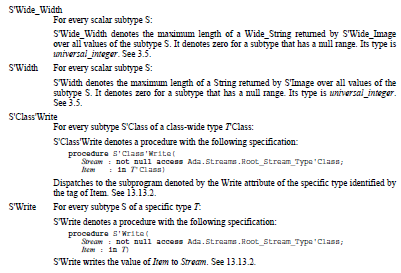






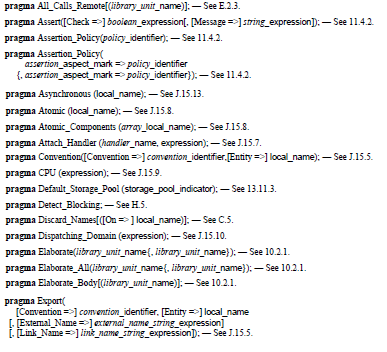


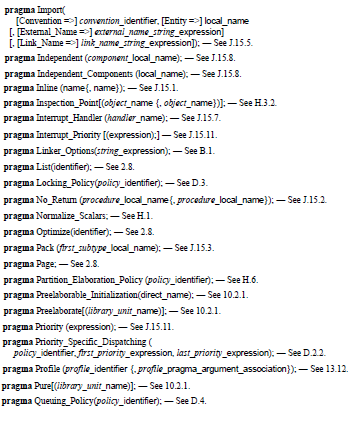


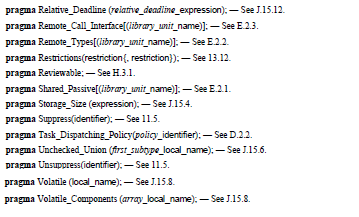


Appendix E

This appendix contains a listing of the language-defined pragmas in Ada. There are cross-references to sections in the 2012 reference manual [1].



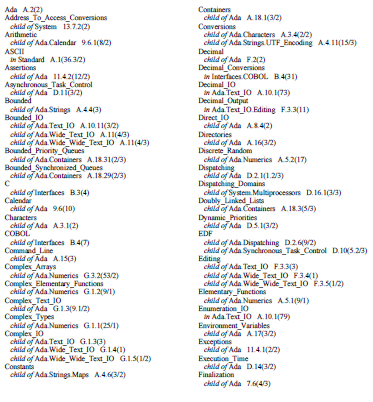




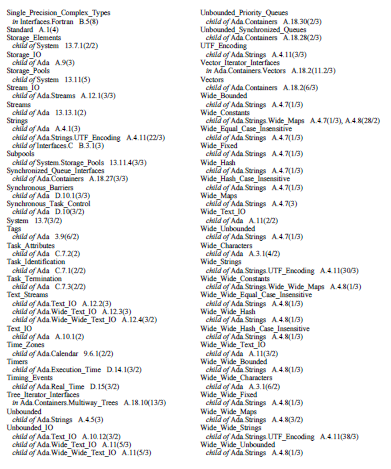
Appendix F

This appendix contains a listing of all language-defined program entities in Ada. This includes packages, types and subtypes, subprograms, exceptions, and objects. There are cross-references to sections in the 2012 reference manual [1].

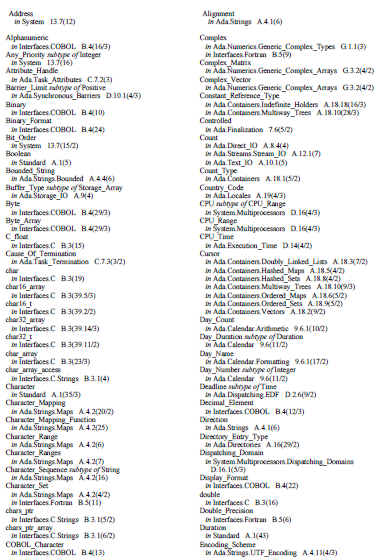
**Language-Defined Packages**

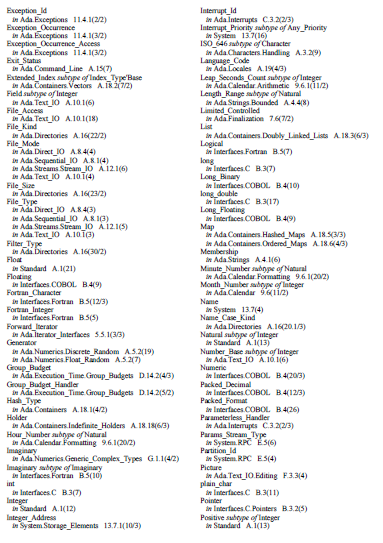


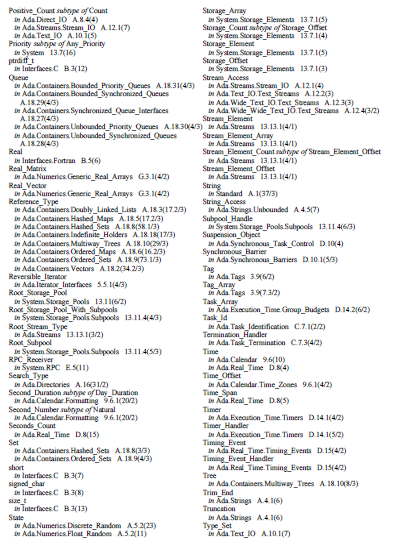




**Language-Defined Types and Subtypes**

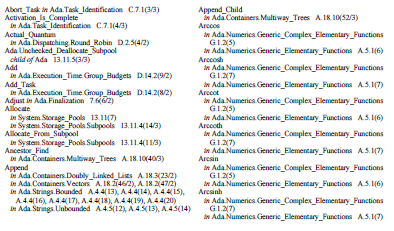


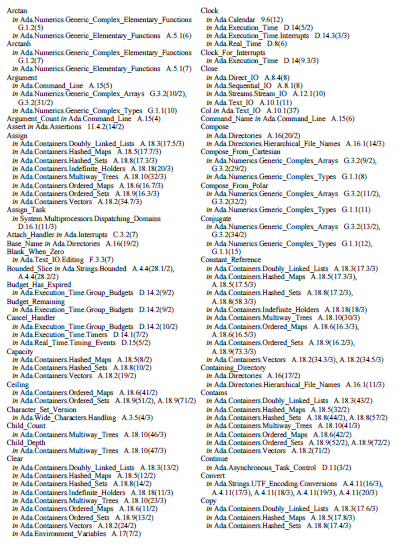


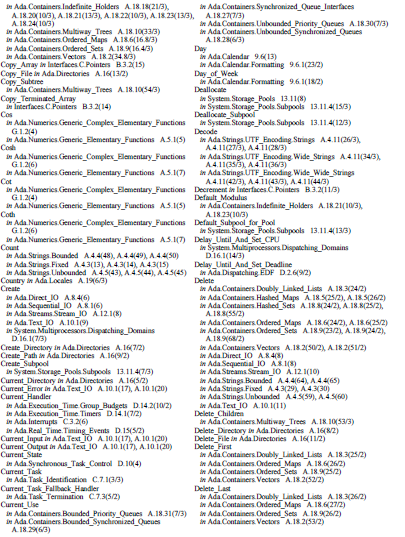


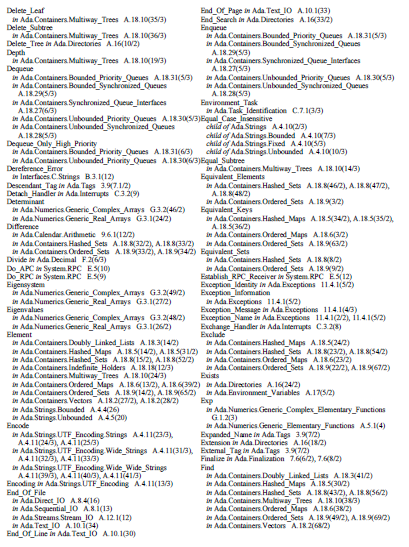


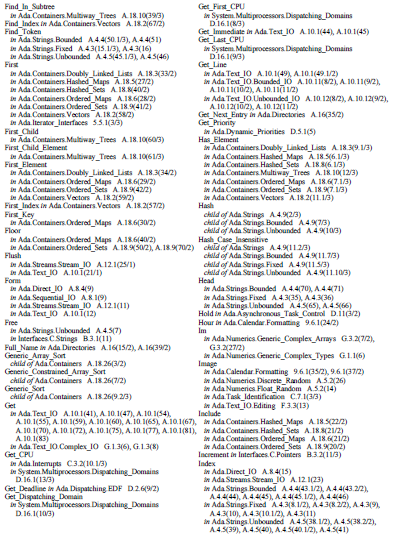
**Language-Defined Subprograms**

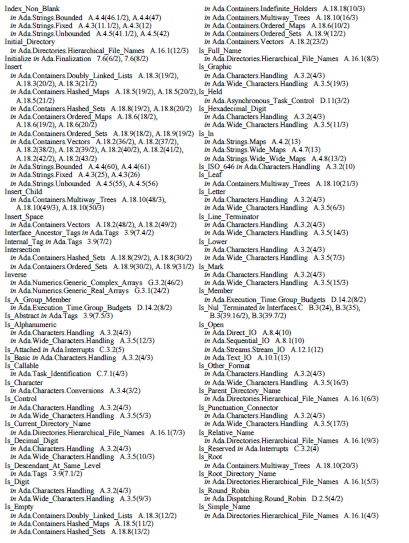


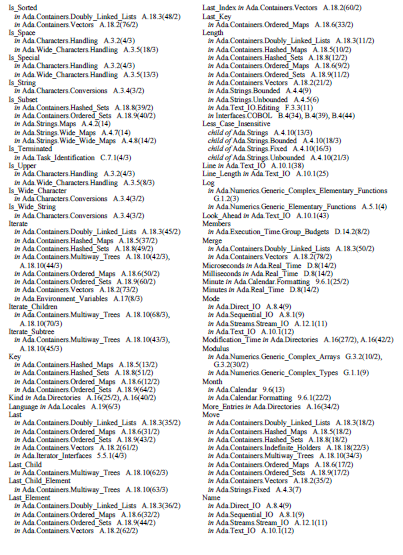


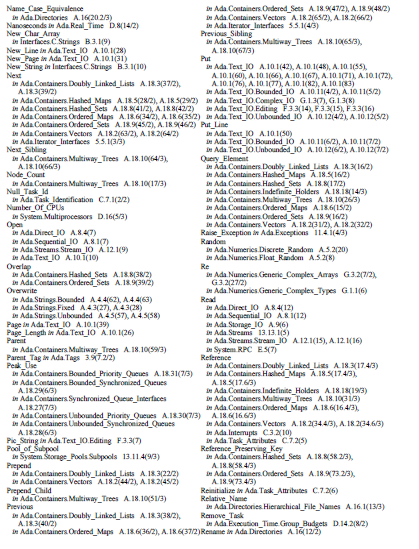


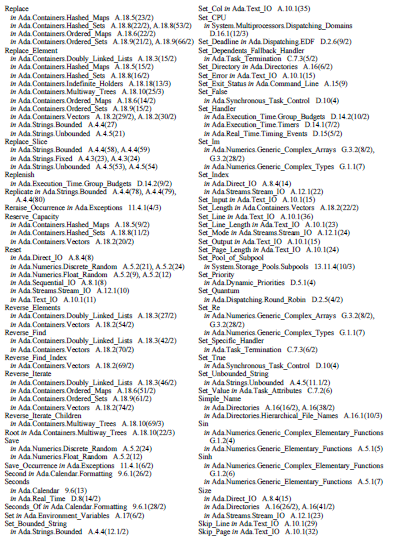


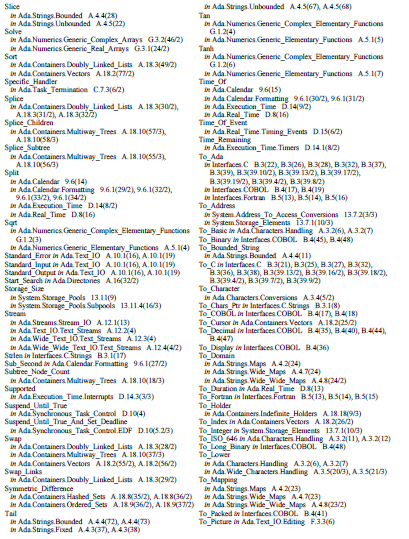


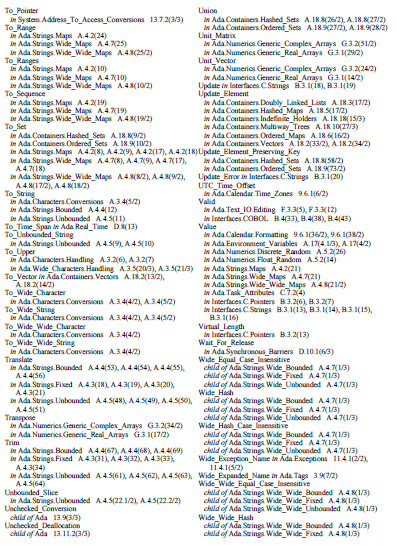
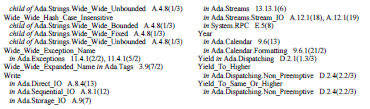








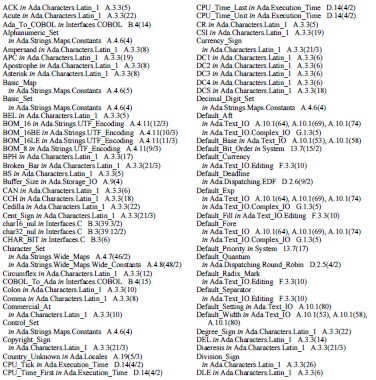


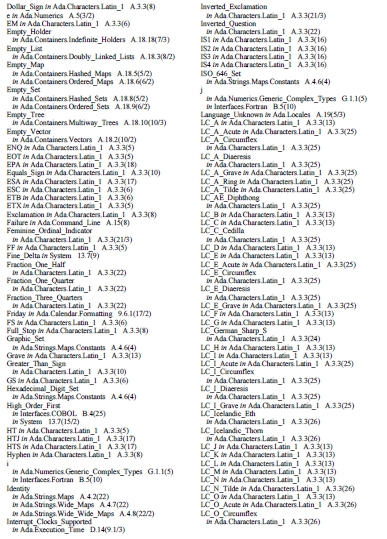


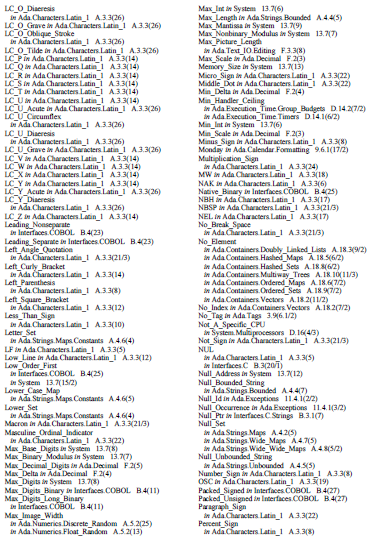
**Language-Defined Exceptions**

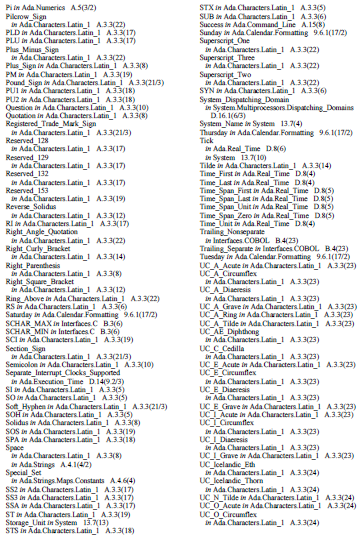


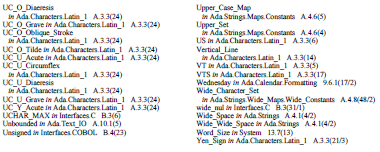
**Language-Defined Objects**





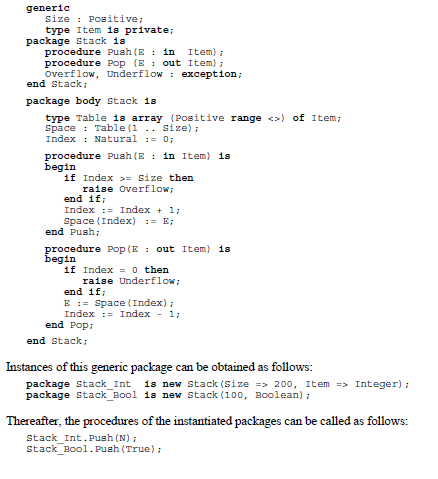






Appendix G

This appendix contains an example of a generic package implementing one possible formulation of stacks. Each stack’s size and element type are provided by generic formal parameters [1].



References

[1] AXE Consultants. (2012). *Annotated Ada reference manual, ISO/IEC 8652:2012(E): Language and standard libraries*. Retrieved from http://www.ada-auth.org/

standards/12rm/RM-Final.pdf

[2] Barnes, J. G. P. (1989). *Programming in Ada*, 3rd ed. Wokingham, England: Addison-Wesley Publishing.

[3] Barnes, J. G. P. (2013). Rationale for Ada 2012: Epilogue. Retrieved from http://www.adacore.com/uploads/technical-papers/Ada2012\_Rationale\_Epilogue-cc.pdf

[4] Brosgol, B. M. (1992). Ada. *Communications of the ACM*, *35*(11), 41-42.

[5] Burns, A., & Wellings, A. (1995). *Concurrency in Ada*. Cambridge, UK: Cambridge University Press.

[6] Feldman, M. B. (2012). Who’s using Ada? Retrieved from http://www.seas.gwu.edu/

~mfeldman/ada-project-summary.html

[7] Freedman, R. S. (1982). *Programming concepts with the Ada language*. New York: Petrocelli Books.

[8] Ichbiah, J., Barnes, J., Firth, R., & Woodger, M. (1991). *Rationale for the design of the Ada programming language*. Cambridge, UK: Cambridge University Press.

[9] IEEE Global History Network. (2011). Past programming languages and their influences on today’s languages and programming paradigms. Retrieved from http://www.ieeeghn.org/

wiki/index.php/Past\_programming\_languages\_and\_their\_influences\_on\_today%27s\_languages\_and\_programming\_paradigms

[10] Pyle, I. C. (1981). *The Ada programming language: A guide for programmers*. London: Prentice Hall International.

[11] Sammet, J. E. (1986). Why Ada is not just another programming language. *Communications of the ACM*, *29*(8), 722-732.

[12] Sebesta, R. W. (2012). *Concepts of programming languages*, 10th ed. Boston: Pearson Education.

[13] Veldema, R. (2011). Principles of programming languages: Imperative languages part 1: Basics [PowerPoint]. Retrieved from https://www2.cs.fau.de/teaching/SS2011/

Principles/2-4.pdf

[14] Watt, D. A. (1990). *Programming language concepts and paradigms*. London: Prentice Hall International.

[15] Wilson, L. B., & Clark, R. G. (1988). *Comparative programming languages*. Wokingham, England: Addison-Wesley Publishing.