Ada 9X Snapshot of Mapping Specification Prior to Revision

Version 4.6
5 June 1992

IR-MA-1250-3

Ada 9X Mapping/Revision Team
Intermetrics, Inc.
733 Concord Avenue
Cambridge, Massachusetts 02138
(617)661-1840

Published by
Intermetrics, Inc.
733 Concord Avenue
Cambridge, Massachusetts 02138

This report has been produced under the sponsorship of the Ada 9X Project Office under contract F08635-90-C-0066.

Acknowledgements

This document was prepared by the Ada 9X Mapping/Revision Team based at Intermetrics, Inc. The members of the team are: W. Carlson, Program Manager; T. Taft, Technical Director; R. Duff (Oak Tree Software); M. Edwards; C. Garrity; R. Hilliard; O. Pazy; D. Rosenfeld; L. Shafer; W. White.

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

This work is continuously being reviewed by the Ada 9X Distinguished Reviewers: E. Ploedereder, Chairman (Tartan); B. Bardin (Hughes); B. Brett (DEC); B. Brosgol (Consultant); N. Cohen (IBM); R. Dewar (NYU); G. Dismukes (Telesoft); A. Evans (Consultant); A. Gargaro (Computer Sciences); M. Gerhardt (ESL); J. Goodenough (SEI); T. Harrison (ISSI); S. Heilbrunner (University of Salzburg: Austria); P. Hilfinger (UC/Berkeley); J. Ichbiah (Consultant: DR emeritus); M. Kamrad II (Paramax Systems); P. Kruchten (Rational); R. Landwehr (CCI: Germany); C. Lester (Portsmouth Polytechnic: UK); D. Luckham (Stanford); L. Månsson (TELIA Research: Sweden); R. Mathis (Consultant); S. Michell (Multiprocessor Toolsmiths: Canada); M. Mills (US Air Force); D. Pogge (US Navy); K. Power (Boeing); O. Roubine (Verdix: France); W. Taylor (Consultant: UK); E. Vasilescu (Grumman).

Other valuable feedback influencing the revision process is also continuously being received from the Ada 9X Language Precision Team (Odyssey Research Associates), the Ada 9X User/Implementor Teams (AETECH, Tartan, Telesoft), the Ada 9X Implementation Analysis Team (New York University) and the Ada community-at-large.

The Ada 9X Project is sponsored by the Ada Joint Program Office. Christine M. Anderson at the Air Force Wright Laboratory, Armament Directorate, is the project manager.

0. Reader’s Guide to the Ada 9X Mapping Specification Snapshot

This ‘‘snapshot’’ of the Mapping Specification is provided for reference purposes only. It represents an initial attempt to incorporate the decisions of the April ’92 ISO WG9 and Distinguished Reviewer meetings. Change bars represent changes relative to version 4.0 of the Mapping Specification.

In some cases, we have not completely fulfilled the recommendations of the WG9 and DR meetings. In these cases, we have included explanatory notes. We intend to use this document as the basis for the revision phase, but these cases and others will of course be again open for comment and refinement during this new phase.

Only chapters 3 to 14 are included.

3. Ada 9X Declarations and Types

3.1. Declarations

A declaration introduces a *view* of an entity (and usually, as in Ada 83, the entity itself). The same entity may have several different views. For example, a private type declaration and the corresponding full type declaration introduce different views of the same type. This term will be defined more precisely during the revision; we use it in this document to explain such things as conversions of actual parameters (see 4.6).

Protected_declaration (see 9.5) is added to the list of basic declarations.

3.2. Objects and Named Numbers

3.2.1. Object Declarations

[*Note*: We were asked to further investigate the issue of parameterized parallel task activation. Here is a possible alternative approach:

To support parameterized parallel task activation, it would be useful to allow an object declaration of an anonymous limited array type to allow an iterator notation, as follows:

```
object_declaration ::=
  ... -- As in Ada 83
 | identifier_list : array(for identifier  in discrete_range
                          {,  for identifier  in discrete_range})
                      of limited_type_mark  discriminant_constraint;
```

Such a declaration would initialize an array of limited components, with the discriminant constraint expressions parameterized by the identifiers appearing in the iterators. This provides more direct support for parallel parameterization than the iterator-in-aggregate mechanism, since parallel activation is already the rule for an array of tasks. By restricting it to limited types, we avoid problems associated with assigning or comparing ragged arrays, while allowing it to be used for arrays of protected objects acting as device drivers, etc.]

3.2.2. Number Declarations

3.3. Types and Subtypes

As in Ada 83, a *type* has an *associated set of values*, and an *associated set of operations*. A *class* is a set of types with similar values and operations, which is closed over derivation. As in Ada 83, types are grouped into *language-defined classes*, such as integer, real, task, etc. In addition, in Ada 9X, types are grouped into *derivation classes*, each formed by the *root* type for the class and its derivatives (direct and indirect).

A *specific type* is a type declared by a type declaration, task type declaration, or protected type declaration. In addition to specific types, there are *class-wide* types (used in conjunction with task types and tagged types, -- see 3.4.1) and *universal* types (used in conjunction with numeric types -- see 3.5.4 and 3.5.6).

A type is either an *elementary* type or a *composite* type. Elementary types are the scalar types and access types. Composite types are the array, record, private, task, and protected types. (Protected types are described in chapter 9.)

[*Notes on terminology*: The term ‘‘specific type’’ corresponds to the normal types in Ada 83. It is introduced to differentiate them from the (new) ‘‘class-wide’’ types and the (preexisting) ‘‘universal’’ types.

In Ada 83, composite types include only record and array types. Here we generalize it to include all types other than scalar and access. The term ‘‘elementary type’’ is new with Ada 9X, to simplify descriptions that in Ada 83 use the phrase ‘‘scalar and access types.’’]

3.3.1. Type Declarations

A type declaration introduces a type and a *first-named* subtype. The name given in the type declaration is the name of the subtype. [*Note*: This is true in many cases for Ada 83; for simplicity, we make it true in all cases for Ada 9X.]

As in Ada 83, a subtype determines a type and a possibly null constraint. An *unconstrained* subtype is one with a null constraint. Given a subtype S for an elementary type, S’BASE denotes the unconstrained subtype of the same elementary type.

In later descriptions, when we refer to properties of a ‘‘type T,’’ we are describing properties that apply to the type in general, rather than to any particular subtype of the type.

[*Note*: Types are never directly named; all type marks denote subtypes. Therefore, when we use the term *anonymous* type we really mean a type that has no nameable subtypes. Anonymous types are declared implicitly as part of certain other declarations, such as single task declarations, single protected declarations, and array and access definitions.]

3.3.2. Subtype Declarations

Floating point and fixed point accuracy constraints are removed from the syntax. A decimal digits constraint may be applied to a decimal fixed point type (see 3.5.9).

```
constraint ::=
   range_constraint
  | index_constraint
  | discriminant_constraint
  | decimal_digits_constraint
```

It is illegal to apply a constraint to a tagged class-wide type (see 3.4.1) or to a generic formal type with (<>) for its discriminant part (see 12.1.2), as they have an unknown set of discriminants.

3.3.3. Classification of Operations

The *primitive* operations of a specific type are the implicitly declared operations of the type, and, for a type declared within a package specification, any subprogram with a formal parameter of the type, result of the type, or access parameter (see 3.9) designating the type, explicitly declared immediately within the same package specification as that of the type declaration. For a protected or task type, the primitive operations also include the protected operations (see 9.7).

The implementation of an implicitly declared primitive operation may be *overridden* by providing a new declaration for the primitive operation, immediately within the same package specification or declarative part as that of the type declaration.

We define *operand matching rules* that determine what operands may be passed to what operations. To simplify description, we say that one type *covers* another if they are the same type, or if the first type is the universal or class-wide type for a class that contains the second type. The *expected* type for an operand determines the types that are allowed for actual operands, as follows:

- If the expected type and the actual type are both specific types, then they must be the same type;
- If the expected type or the actual type is a universal type then one of the types must cover the other;
- If the expected type is a class-wide type, then it must cover the actual type;
- If the expected type is a specific type, and the actual type is a class-wide type, then the operand must be a controlling operand (see 3.4.2) and the class-wide type must be for the class rooted at the expected type.

If the formal is an access parameter (see 3.9), then the actual *operand* is the object designated by the value of the actual parameter, and the expected type of that operand is the designated type of the access parameter.

Class-wide and universal types do not have their own primitive operations, nor any implicitly declared operations other than attributes. Because of the operand matching rules, the primitive operations of the root type for a class may be used for operands of the corresponding class-wide or universal type.

A subprogram with a formal parameter or result of a class-wide type may be explicitly declared. Such subprograms are referred to as *class-wide* operations.

For an elementary subtype S, the attribute S'BASE is allowed to appear wherever a type mark may appear, rather than strictly as a prefix for other attributes.

3.4. Derived Types

The syntax for a derived type definition is amended to include an optional type extension part (see 3.4.1):

derived_type_definition ::=
  new subtype_indication [type_extension_part]

A type extension part is allowed only for tagged types (see 3.4.1).

The description of the characteristics of a derived type is revised as follows:

- The value set of a derived type is a (possibly modified) copy of the parent's value set. The copy is modified in these cases:
  - If the derived type declaration includes a discriminant part (see 3.6.1), then the parent subtype must be constrained, and the discriminants of the derived type are those given in the new discriminant part. Otherwise, as in Ada 83, the derived type has the same discriminants as the parent type, if any. [*Note*: The constraint on the parent may depend on the new discriminants — see 3.6.1.]
  - If the derived type declaration includes a record extension part (see 3.4.1), then the

parent subtype must be constrained, and the values consist of the discriminants (if any), followed by the non-discriminant components of the parent type, followed by the components of the record extension part.

- The derived type *inherits* the primitive operations of the parent type that are declared prior to the derivation, obtaining the specification of these operations by systematic replacement of the parent type with the derived type, where it appears as the type of a formal parameter or result (as in Ada 83 — see RM 3.4(13)), or as the designated type of an access parameter (see 3.9).
- If an explicitly declared primitive operation of a derived type is a homograph of an inherited operation, then the explicitly declared operation *overrides* the inherited one.
- As in Ada 83, when STANDARD.BOOLEAN is the parent type, the result type is not replaced when inheriting the predefined relational operators and membership tests; the result type remains STANDARD.BOOLEAN for these operations of the derived type (RM 3.4(21)).

For a composite type, the *parent part* consists of all components inherited from the parent type, and the *extension part* consists of any other components.

The parent type in a derived type definition must not be a class-wide type, nor an unconstrained generic formal type with an unspecified set of discriminants (see 12.1.2).

3.4.1. Tagged Types and Type Extensions [*new*]

A *tagged* type is a record or private type that provides a type *tag* that represents, at run time, the identity of the type. Tagged types support type extension (see below). A record or private type is tagged if its declaration includes the reserved word **tagged**, or if it is derived from a tagged type.

The derived type definition for a type derived from a tagged type must include a type extension part:

```
type_extension_part ::=
   with record_type_definition
   | with private
   | with null
```

A tagged type must be extended with either a record extension part (**with** record_type_definition -- see 3.8.2), private extension part (**with private** -- see 7.4.2), or null extension part (**with null**). A derived type defined in one of these ways is called a *type extension* of its parent type. [*Note*: By requiring **with null** we ensure that all tagged types include either the reserved word **tagged** or **with** in their declaration. Furthermore, it emphasizes that normal type conversion cannot be used to construct a value of the derived type given a value of the parent type, but instead an extension aggregate using **with null** must be used (see 4.3.1). This should enhance readability (at some expense of writability, though we expect null extensions to be less common than record extensions.]

Example of declaration and extension of a tagged type:

```
    type VEHICLE is tagged
      record
        WEIGHT : KILOGRAMS;
      end record;

    type CAR is new VEHICLE with
      record
        NUMBER_OF_DOORS    : POSITIVE range 1..5;
        NUMBER_OF_WINDOWS  : POSITIVE range 4..8;
        NUMBER_OF_WHEELS   : POSITIVE range 3..4;
      end record;
```

When a tagged type T is declared, an associated class-wide type, denoted by T'CLASS, is implicitly declared. The value set for T'CLASS is the union of the value sets of the specific types in the derivation class rooted at T. The union is a discriminated union, with the type tag as the (implicit) discriminant.

Given a tagged subtype S for the type T, S'CLASS denotes a subtype of T'CLASS. A value of type T'CLASS belongs to S'CLASS if it satisfies the constraints applicable to S.

If a tagged type's parent is not limited, then any extension part must not contain any limited components. The scope level (see 8.2) of a derived tagged type declaration must be the same as that of its parent. [*Note*: Thus, all types included in a tagged class are at the same scope level.]

A primitive operation of a derived tagged type that overrides an inherited operation must be subtype conformant (see 6.3.1) with the inherited operation.

For a tagged type, the declaration of a primitive operation must not occur after a forcing occurrence (see RM 13.1(6)) of the type. [*Note*: This restriction minimizes confusion by ensuring that the implementation reached via a dispatching call corresponds to the operation declared immediately within the declarative region where the type is declared. This also implies that all overridings for a tagged type must be declared prior to subprogram bodies, even for tagged types declared in a package body.]

The declaration of a tagged derived type with a record or null extension part is considered a forcing occurrence for the parent type. [*Note*: This combined with the previous rule ensures that all derivatives of a tagged type inherit all of its primitive operations, so that dispatching will work properly.]

Each tagged specific type T has a tag value, which represents the identity of the type at run time. A tag value uniquely identifies a particular tagged type declaration. Repeated elaborations of the same declaration need not produce distinct tag values, nor do repeated instantiations of the same generic body. However, distinct instantiations of the same generic specification containing a declaration for a tagged type must produce distinct tag values.

Each value of a tagged class-wide type has a tag, which reflects the specific type from which the value originated, possibly through one or more (view) conversions (see 4.6).

Each object of a tagged class-wide type must be explicitly initialized, and is constrained thereafter to hold only values with tag and any discriminants matching those of its initial value.

3.4.2. Operations of Tagged Types [*new*]

By definition, a primitive operation (see 3.3.3) of a type T has a formal parameter of type T, return type T, or an access parameter (see 3.9) designating T. The actual parameter passed to a formal parameter of type T is called a *controlling operand* of the primitive operation. The object designated by the actual parameter passed to an access parameter designating T is also a controlling operand. If the return type is T, the result of the operation is called a *controlling result* of the primitive operation.

For a tagged type, the primitive operations are called *dispatching* operations, because when applied to a controlling operand that is class-wide, they automatically *dispatch* to the appropriate implementation of the operation.

A call to a dispatching operation of a tagged type T works as follows:

- If there are no class-wide controlling operands anywhere in the context that determine the tag (as defined by the next bullet), then the controlling tag value is by definition that for T. (this is a normal ‘‘static’’ binding case).
- Otherwise, each controlling operand determines a tag:
    - If the operand is class-wide, the tag is that of the operand’s value;
    - If the operand is a controlling result of a dispatching operation, the tag is the same as the controlling tag value of that operation;
    - Otherwise, the tag is that of the specific type of the operand (which, according to the matching rules, must identify T, i.e. this is a ‘‘static’’ binding case).

  If all controlling operands determine the same tag value, then this is the *controlling tag value* for the operation. Otherwise, CONSTRAINT_ERROR is raised, or, in the case of an equality operator ("=" or "/=") whose result is of type BOOLEAN, inequality is indicated.
- The controlling tag value determines the particular implementation of the operation to be invoked. The implementation selected is the one for the corresponding primitive operation of the specific type identified by the tag.

If a dispatching operation has a controlling result, but has no operand that determines a tag, then its controlling tag is chosen to match that required by its context, in order to avoid CONSTRAINT_ERROR.

If a single operation is a primitive for more than one tagged type, then in a given call, only one of the types may have actual controlling operands that are class-wide. [*Note*: We shifted this check to the call rather than the declaration, to handle situations where an operation is declared on two private types that both turn out to be tagged in their full type declarations.]

[*Note*: We are also considering disallowing calls where some of the controlling operands of a given type are class-wide, while others have statically-determined tags. Such calls seem to be a possible source of confusion.]

If a tagged type has any primitive operations that are declared as abstract (see 6.1), then it is called an *abstract type*, and it is illegal to create an object of that type, and a value of the type must only be produced via a conversion (see 4.6). [*Note*: This disallows, for example, object declarations, allocators, aggregates, using the type as a generic actual, and non-abstract functions returning the type.]

When calling an abstract primitive operation of a tagged type T, the controlling tag value must not be that of type T. [*Note*: This check can be performed at compile-time, because there are no values of the class-wide type T’CLASS with tags identifying type T.]

When deriving from an abstract type, all primitive operations that are abstract for the parent type must be overridden for the derived type, possibly with another abstract subprogram declaration (see 6.1). As in Ada 83, an inherited operation with a result of the derived type is defined in terms of a conversion of the result returned by the parent type's operation. If the parent type is tagged, conversion to the derived type is not defined (see 4.6), so the inherited operation must be overridden, just as though it had been an abstract operation of the parent type.

Note:

All primitive operations of a tagged type are dispatching, whether inherited, defined by an instantiation, a renaming declaration, or a normal subprogram declaration. On the other hand, non-primitive operations are never dispatching, even when they are defined by renaming a dispatching operation.

3.5. Scalar Types

3.5.1. Enumeration Types

3.5.2. Character Types

The predefined type CHARACTER will be changed to have 256 positions, the first 128 of which will correspond to ASCII. A new predefined type WIDE_CHARACTER will be provided, to support character sets with more than 256 positions, in accordance with the recommendations of the CRG (see chapter 2).

All character literals are directly visible throughout their scope. The type of a character literal must be determinable solely from context, but using the fact that the literal is a value of some character type. The character literal must be one of the character literals of the type (although this is not used to determine the type of the literal). [*Note*: This is to be consistent with other ‘‘unrenamable’’ literals: string literals; null; and the numeric literals (see RM 4.2(4)). Since there will be two character types in scope at all times (CHARACTER and WIDE_CHARACTER), it is simpler to always use context to resolve the type of a character literal than to try to use the particular character itself to determine the type.]

3.5.3. Boolean Types

3.5.4. Integer Types

By definition, all integer types are derived (directly or indirectly) from *root_integer*. In addition, a universal type, *universal_integer*, is defined for the integer class. Integer literals and integer named numbers are defined to yield a value of this universal type. The operand matching rules (see 3.3.3) allow a universal integer to be used as an actual operand if the expected type is some specific integer type.

The VAL attribute is defined to have an operand of the universal type *universal_integer*, which allows an actual operand of any integer type.

Values of *universal_integer* may be combined using the predefined operators of any specific integer type. However, in the case of ambiguity, there is a preference for the operators of the *root_integer* type (see 8.7). [*Note*: The operators of *root_integer* correspond to the predefined operators of *universal_integer* in

Ada 83. The result of such an operator is never universal. This eliminates the need for the special case rules governing ‘‘convertible universal operands’’ given in RM 4.6(15).]

The value set for all integer types is the (infinite) set of mathematical integers. For static calculations with the operators of *root_integer*, an implementation must support all such values (see 4.9). For run-time calculations, an implementation may restrict integer values to a finite range. The FIRST and LAST attributes of an unconstrained integer subtype specify the minimum range supported by the implementation for run-time operations on values of the corresponding integer type. At run-time, a predefined operation taking or yielding an integer value outside of this minimum range either produces the correct mathematical result, or raises CONSTRAINT_ERROR.

As in Ada 83, the FIRST and LAST attributes of a constrained integer subtype represent constraints on the value of any object of the subtype.

The run-time range of the *root_integer* type is at least SYSTEM.MIN_INT .. SYSTEM.MAX_INT (see RM 13.7.1).

STANDARD.INTEGER’LAST is at least 2**15-1. If an implementation provides STANDARD.LONG_INTEGER, then the value of its LAST attribute is at least 2**31-1.

As in Ada 83, an integer type definition selects a parent type from one of those defined in package STANDARD. These types are called the *standard integer types*. An implementation may also provide additional *non-standard* integer types declared outside package STANDARD. Such types may have a run-time range outside of SYSTEM.MIN_INT .. SYSTEM.MAX_INT, and need not be symmetrical around zero. An implementation may restrict the use of such types; in particular, they need not be allowed as array or loop indices. [*Notes*: Non-standard integer types may not be used as generic actual types for a formal integer type or formal discrete type -- only for formal private and formal derived types. The System Programming Annex defines requirements for unsigned types to be provided in package SYSTEM.UNSIGNED_SUPPORT. Such types are considered non-standard integer types.]

3.5.5. Operations of Discrete Types

3.5.6. Real Types

The syntax is amended to remove accuracy constraints, as explained in section 3.3.2.

```
real_type_definition ::=
  floating_point_definition | fixed_point_definition
```

By definition, all real types are derived from *root_real*. In addition, a universal type *universal_real* is defined for the real class of types. Real literals and real named numbers are of type *universal_real*, as are the results of certain attributes.

Values of type *universal_real* may be combined using the predefined operators of any specific real type. However, in the case of ambiguity, there is a preference for the operators of *root_real* (see 8.7).

The value set for all real types is the infinite set of rational numbers. For static calculations with the operators of *root_real*, an implementation must support all such values. For run-time calculations, an implementation may use approximations and may restrict the range. The attributes of a real type define the maximum relative (floating-point) or absolute (fixed-point) error and minimum range supported by the implementation for representing and operating on run-time values. If the MACHINE_OVERFLOWS attribute is true, then at run-time, a predefined operation taking or yielding a real value outside of this minimum range either produces the correct result (as defined by the accuracy model — see NM:ALL), or raises CONSTRAINT_ERROR.

As in Ada 83, the FIRST and LAST attributes of a constrained real subtype represent constraints on the value of an object of the subtype.

At run-time, the *root_real* type is a floating point type, and its DIGITS attribute is at least SYSTEM.MAX_DIGITS (see RM 13.7.1). The LARGE attribute for *root_real* is at least as great as any other floating point type declared in STANDARD with DIGITS equal to SYSTEM.MAX_DIGITS.

STANDARD.FLOAT'DIGITS is at least 6. If an implementation provides STANDARD.LONG_FLOAT, then the value of its DIGITS attribute is at least 11.

In accordance with AI-00174, the values of the FIRST and LAST attributes of a real type depend on the representation chosen for the type. Therefore, a range constraint on a real type is a forcing occurrence (RM 13.1(6)) for the type. [*Note*: This eliminates certain anomalies relating to the fact that determining whether a range constraint is compatible with the real type requires selecting the representation for the type.]

As in Ada 83, a floating point or fixed point type definition selects a parent type from one of those defined in package STANDARD. These types are called the *standard real types*. An implementation may also provide additional *non-standard* real types declared outside package STANDARD. Such types may have a DIGITS attribute greater than SYSTEM.MAX_DIGITS, or a mantissa greater than SYSTEM.MAX_MANTISSA. An implementation may restrict the use of such types; in particular, they need not have a full set of attributes. [*Notes*: Non-standard real types may not be used as generic actual types for a formal floating point type or formal fixed point type.]

3.5.7. Floating Point Types

The syntax is amended to remove accuracy constraints, as explained in section 3.3.2.

> floating_point_definition ::=
>   **digits** *static*_simple_expression [real_range_specification]
>
> real_range_specification ::=
>   **range** *static*_simple_expression .. *static*_simple_expression

3.5.8. Operations of Floating Point Types

3.5.9. Fixed Point Types

The syntax is amended to remove accuracy constraints, as explained in section 3.3.2.

```
fixed_point_definition ::=
   delta static_simple_expression real_range_specification
 | decimal_fixed_point_definition

decimal_fixed_point_definition ::=
 delta static_simple_expression
   digits static_simple_expression [real_range_specification]

decimal_digits_constraint ::=
 digits static_simple_expression [range_constraint]
```

A fixed point type declared with both a **delta** and a **digits** is a *decimal fixed point* type. The specified delta must be a power of 10, and the *small* is defined to equal the delta.

The DIGITS attribute of a decimal fixed point subtype is determined by the decimal digits constraint that applies to it. The FIRST and LAST attributes of a decimal fixed point subtype are determined by the range constraint, if any, that applies to it. In the absence of a range constraint, the FIRST and LAST attributes are determined by the applicable digits constraint, such that LAST = *delta**(10.0***digits*-1.0) and FIRST = -LAST.

Notes:

For implementations that conform to the Information Systems Annex, *small*s that are a power of 10 must be supported, as well as at least 18 digits for a decimal fixed point definition. See IS:ALL.

3.5.10. Operations of Fixed Point Types

3.6. Composite Types [*new*]

Composite types may have *components*, including special components called *discriminants*.

The rules regarding declarations of a variable of a composite type are relaxed to conform to the rules regarding constant declarations, namely that if the subtype indication is for an unconstrained subtype with undefaulted discriminants or unconstrained array bounds, then there must be an initialization expression, and the constraints of the variable are defined by the initial value.

3.6.1. Discriminants [*originally RM 3.7.1*]

Discriminants represent a general method for parameterizing types. The syntax for discriminant specifications is modified to allow an *access discriminant*, with a type specified by an access definition (see 3.9):

```
discriminant_specification ::=
   identifier_list : type_mark [:= expression]
 | identifier_list : access_definition [:= expression]
```

A discriminant part may only be specified in a type declaration for a composite type that is not an array type. The type of an access discriminant is an anonymous general access type (see 3.9). The type of other discriminants must be discrete.

Only a limited type may have an access discriminant.

If a discriminant part is specified in a derived type declaration, then the parent subtype must be constrained.

Analogous to the Ada 83 rule for component constraints (RM 3.7.1(6)), when a discriminant is used in a constraint on the underlying type, it must appear alone (not as part of a larger expression). In addition, if discrete, its subtype must be statically compatible with the underlying discriminant subtype (see 4.9.1). [*Note*: This restriction minimizes the checking required on type conversion. See 4.6.]

For a tagged type, the discriminants specified in a derived type declaration need not be used to constrain the parent subtype.

Within the definition of a type with a discriminant part, the name of a discriminant may be used in an expression. However, as in Ada 83, a discriminant may not be used in a component's range constraint, and when used in other constraints, it must be used alone, not as part of a larger expression. Within a task or protected body, a discriminant acts as a formal **in** parameter to the unit.

3.6.2. Discriminant Constraints [*originally RM 3.7.2*]

3.7. Array Types [*originally RM 3.6*]

3.7.1. Index Constraints and Discrete Ranges [*originally RM 3.6.1*]

The rules for determining the type of a discrete range are revised so that if the type for both bounds is either *root_integer* or *universal_integer*, then the type for the range is STANDARD.INTEGER. [*Note*: This resolves the Ada 83 problem where ‘‘**for** I **in** -1 .. 100’’ is ambiguous.]

3.7.2. Operations of Array Types [*originally RM 3.6.2*]

3.7.3. String Types [*originally RM 3.6.3*]

An additional predefined string type WIDE_STRING will be provided, defined as an array of WIDE_CHARACTERs indexed by POSITIVE, in accordance with the recommendations of the CRG (see chapter 2).

3.8. Record Types [*originally RM 3.7*]

A type definition for a record type may include the reserved words **tagged** and **limited**:

```
type_definition ::=
    . . .  as in Ada 83
  | [tagged] [limited] record_type_definition
```

If the reserved word **tagged** appears, the record type is a tagged type, and may be extended in a type derivation (see 3.4.1). If the reserved word **limited** appears, the record type is limited throughout its scope (that is, equality and assignment are not predefined — see 7.4.5).

[*Note*: The concept of an explicitly limited record type has been retained pending further review.

Because finalization, limited type extension, and access discriminants are all dependent on the use of limited types independent of privateness, we prefer to retain the ability to specify limitedness without specifying privateness.]

[*Note*: The subsections on Discriminants have been moved to 3.6.1 and 3.6.2.]

3.8.1. Variant Parts [*originally RM 3.7.3*]

[*Note*: As in Ada 83, a discriminant controlling a variant part must be discrete.]

3.8.2. Record Extension [*new*]

A tagged record or tagged private type may be extended with a record extension part (see 3.4.1).

Within the extension part, the value of an access discriminant (see 3.6.1) of a component may be specified by an ACCESS attribute whose prefix is the simple name of the derived type being declared. When used in this way, the attribute designates the enclosing object being initialized. [*Note*: This allows a form of ‘‘multiple inheritance,’’ by embedding a component with an access discriminant that designates the enclosing object.]

[*Note*: The Ada 83 rules for naming of record components (RM 3.7(3)) imply that overriding the components of a visible record type in the extension part is illegal. However, in a record extension of a private type, a component in the extension part may have the same name as a component of the private parent part; as in Ada 83, the parent part remains private throughout the scope of the type extension.]

3.8.3. Operations of Record Types [*originally RM 3.7.4*]

3.9. Access Types [*originally 3.8*]

The syntax for access type definitions is changed to support general access types (including access-to-constants) and access-to-subprograms.

```
access_type_definition ::=
   access [general_access_modifier] subtype_indication
 | access [protected] procedure [formal_part]
 | access [protected] function [formal_part] return type_mark

general_access_modifier ::= all | constant
```

An access value is either **null**, or designates some entity. Access-to-object values designate objects. Access-to-subprogram values designate subprograms.

There is a *storage pool* associated with each access-to-object type. An allocator for an access-to-object type allocates in that type’s pool.

An access type with a general access modifier is a *general access type*. Values of a general access type may be produced by allocators, by conversion from other access types (see 4.6), and by use of the ACCESS attribute (see 3.9.2). If the modifier is the reserved word **constant**, then the designated object may only be read through a value of such a type. If the modifier is the reserved word **all**, then the designated object may be both read and updated through a value of such a type.

Other access-to-object types are *pool-specific access types*, which correspond to the access types of Ada 83. Values of such types may only be produced by allocators (and, as in Ada 83, by conversion from an access type derived from some common ancestor).

An *access definition* may be used to specify a formal **in** parameter (see 6.1) or a discriminant (see 3.6.1):

access_definition ::= **access** type_mark

Such a parameter or discriminant is of an *anonymous access type*. An anonymous access type is a general access-to-variable type, implicitly declared at the point of the access definition. Its designated subtype is that denoted by the type mark. An anonymous access type has no primitive operators of its own. [*Note*: An access discriminant or parameter may be converted (see 4.6), used as a prefix of a name (see RM 4.1(8)), used to initialize another access discriminant, or passed as an actual parameter where the formal is an access parameter (see 3.3.3).]

[*Notes*:

The language rules (see 3.9.2 and 4.6) ensure that (unless the unchecked programming features of Chapter 13 are used) a pool-specific access value will always designate an object allocated from its own pool. By contrast, a general access value can designate an object allocated from any pool, or created by an aliased object declaration, so long as the object is of an appropriate subtype and scope level.

Which pool is associated with an access type is implementation defined, if not user specified. Sections 13.2 and SP:ALL define methods for specifying the storage pool of an access type, and for specifying various aspects of the pools themselves. Several access types may share the same pool.

Access parameters support tag-based dispatching on access values (see 3.4.2). They also provide a way to pass a reference to an aliased local variable to a subprogram. All objects of an anonymous access type are constants. The object designated by an access discriminant or parameter is a variable. A value of an anonymous access type is never **null**. A run-time check is made when the value of an access parameter or discriminant is specified to ensure that it is not null (see 4.6). A run-time scope check is performed when an access parameter is converted to another general access type, to ensure that it is not designating an object at a deeper scope-level than that of the resulting access type (see 4.6).

Pools and the terms ‘‘general’’/‘‘pool-specific’’ are not defined for access-to-subprogram types.

There is no requirement that the ACCESS attribute of a particular subprogram always yield the same value. The value of the attribute may designate, for example, a locally generated interface routine needed to support an indirect call.

End of Notes.]

3.9.1. Incomplete Type Declarations [*originally 3.8.1*]

3.9.2. Operations of Access Types [*originally 3.8.2*]

General access types may only designate *aliased* objects. The object created by an allocator is an aliased object. All objects of a limited tagged type, or non-private limited type, are *inherently aliased*. [*Note*: We would have made all limited types inherently aliased, except that a limited private type may be completed with an elementary type, which can't be passed by reference. Inherent aliasing is important for parameter passing and function return of task types, protected types, controlled types, types with access discriminants, and types composed of such types (see 7.4.5). This issue will receive further study during the revision phase.] An object or component with the reserved word **aliased** in its object or component subtype definition is aliased. The following new syntax is provided for this purpose:

```
object_declaration ::=
   identifier_list : [constant] object_subtype_definition
     [:= expression];

object_subtype_definition ::=
   [aliased] subtype_indication
  | [aliased] array_type_definition

component_subtype_definition ::= [aliased] subtype_indication

unconstrained_array_definition ::= ... of component_subtype_definition

constrained_array_definition ::= ... of component_subtype_definition
```

An aliased object with discriminants is constrained by the initial value of its discriminants, whether created by an allocator or by a declaration. [*Note*: For allocators, this corresponds to the rule in Ada 83. The rule is necessary to support access subtypes safely.]

For a type T, an allocator **new** T is overloaded on all access types designating types that cover T. If T is a tagged class-wide type, then the allocator must specify an initial value.

For an aliased object X of type T, X’ACCESS is overloaded on all general access types designating types that cover T.

For X’ACCESS, the constraints of X must statically match (see 4.9.1) the constraints on the designated subtype of the result type, unless it is an unconstrained subtype with discriminants. In Addition, the same representation clauses and pragmas must apply to X that apply to the designated subtype. Finally, X must not be a subcomponent that depends on discriminants of a variable, if the discriminants have defaults, and the variable might be unconstrained (see RM 8.5(5)). If X is constant, then the type of X’ACCESS must be an access-to-constant type. The scope level of X must not be deeper than that of the resulting access type.

The attribute S’ACCESS, when applied to a prefix S that denotes a subprogram, creates an access value that designates the subprogram.

For an overloaded subprogram name S, S’ACCESS is overloaded on all access types designating subprograms. The context must uniquely determine such a type. The access type determined may then be used to identify the particular subprogram denoted by S. [*Note*: This use of context to resolve the overloading of an attribute prefix is new to Ada 9X. It is not permitted in Ada 83 — see RM 4.1.4(3).]

For S’ACCESS, the scope level of S (see 8.2) must not be deeper than that of the resulting access type. S must be subtype conformant (see 6.3.1) with the subprogram formal part and return subtype specified in the access type definition. S must be a protected subprogram, if and only if the definition of the resulting access type includes the reserved word **protected**. S must not be an abstract subprogram (see 6.1).

The reserved word **all** is used for dereferencing all access types. The ‘‘**.all**’’ may be omitted when used as a prefix, or (for access-to-subprogram) followed by an actual parameter part. Any selected component with a value of an access-to-constant type as prefix is constant. [*Note*: Other access values might provide views of the same object; those views might be variable.]

When an access-to-subprogram value is dereferenced, the resulting subprogram is never a dispatching subprogram.

3.10. Declarative Parts [*originally 3.9*]

The ordering requirement between basic declarative items and later declarative items is eliminated. The occurrence of a body forces (see RM 13.1(6)) the determination of the representation of all types declared prior to the body.

The syntax for declarative parts is modified as follows:

```
declarative_part ::= {declarative_item}

declarative_item ::=
   basic_declaration | representation_clause | use_clause | body

body ::= proper_body | body_stub

proper_body ::=
   subprogram_body | package_body | task_body | protected_body
```

4. Ada 9X Names and Expressions

4.1. Names

4.1.1. Indexed Components

4.1.2. Slices

4.1.3. Selected Components

4.1.4. Attributes

4.2. Literals

4.3. Aggregates

4.3.1. Record Aggregates

A record aggregate for a type must not be used in a place where any extension part of the type is private (see 3.4.1).

A special form of record aggregate, an *extension aggregate*, may be used to specify a value for a tagged derived type in terms of a value of its parent type, as follows:

```
record_aggregate ::=
   aggregate -- as before
 | extension_aggregate

extension_aggregate ::=
   (expression  with record_aggregate)
 | (expression  with null)
```

[*Note*: This feature replaces the concept of *constructor conversions* that appeared in earlier mapping specifications.]

The derived type must be a record extension or null extension of the parent type. The expression must be of the parent type. The nested record aggregate, if present, must specify a value for each discriminant and each component of the record extension part of the derived type. If **with null** is specified, the derived type must be a null extension of the parent type with all discriminants inherited from the parent type.

For the evaluation of an extension aggregate, the expression of the parent type and the expressions for the components in the nested record aggregate are evaluated in an order based on the rules for normal record aggregates (see RM 4.3.1 and AI-00189). [*Note*: This generally means in any order, except for nested array aggregates or string literals whose bounds are determined by a discriminant.] A check is made that the values specified by the aggregate for the parent part and each component of the extension part belong to the corresponding parent or component subtype, CONSTRAINT_ERROR is raised if this check fails.

4.3.2. Array Aggregates

Paragraph 6 of RM 4.3.2, which deals with aggregates having both an **others** choice and other named associations, is removed. The presence of other named associations has no effect on the legality of the use of an **others** choice in an array aggregate. [*Note*: This is possible by stipulating that no implicit array subtype conversion (see 4.6) takes place on an array aggregate with an **others** choice.]

4.4. Expressions

The restriction on the use of **out** parameters is eliminated. [*Note*: Prior to being assigned, **out** parameters are essentially like variables declared without an explicit initialization.]

4.5. Operators and Expression Evaluation

4.5.1. Logical Operators and Short-circuit Control Forms

4.5.2. Relational Operators and Membership Tests

For a membership test of the form ‘‘X **in** S’’ where S is a subtype of type T, the type of X must cover or be covered by T. The result is TRUE if and only if X satisfies the constraints associated with subtype S, if any, and, if X is of a tagged class-wide type, the tag of X identifies a (specific) type covered by T.

[*Note*: If X is of a tagged class-wide type, and T is a specific type, then X **in** T asks whether X’TAG = T’TAG, whereas X in T’CLASS asks whether X’TAG is the tag for T or any of its derivatives.]

4.5.3. Binary Adding Operators

4.5.4. Unary Adding Operators

4.5.5. Multiplying Operators

Corresponding to the Ada 83 operators on *universal_real*, *universal_integer*, and *universal_fixed*, in Ada 9X, the following additional operators appear in package STANDARD for the root numeric types:

```
function "*"(LEFT : root_real; RIGHT : root_integer)
  return root_real;
function "*"(LEFT : root_integer; RIGHT : root_real)
  return root_real;
function "/"(LEFT : root_real; RIGHT : root_integer)
  return root_real;
```

The *root_integer* and *root_real* types also have the full set of operators defined in Ada 83 for integer and real types, respectively. Note that, the above operators on *root_integer* and *root_real* are not predefined for other numeric types.

As in Ada 83, any two fixed point types may be multiplied or divided. In addition, for Ada 9X, we allow either of the operands to be of type *universal_real*.

```
function "*"(LEFT, RIGHT : any_fixed_or_universal_real)
  return precise_fixed;
function "/"(LEFT, RIGHT : any_fixed_or_universal_real)
  return precise_fixed;
```

Like the Ada 83 type *universal_fixed*, the type *precise_fixed* is a real type with no operators, and must be converted before further use.

[*Note*: In response to the April ’92 DR meeting, we are investigating allowing multiply and divide for fixed-point types to be used without explicit conversion of the result, where the type of the result can be determined unambiguously from context. This would treat these fixed-fixed operations roughly the same way that **null** or string literals are treated, requiring the result type of the multiply or divide to be determined from context knowing only that it must be a fixed-point type. The simplest formulation is as follows:

```
function "*"(LEFT, RIGHT : universal_fixed)
  return universal_fixed;
function "/"(LEFT, RIGHT : universal_fixed)
  return universal_fixed;
```

Universal fixed would cover all fixed point types (see 3.3.3 for definition of "cover"), but there would be no predefined operators other than "*" and "/" that would operate on it directly. In this way, it would be like a literal without any preference rule, requiring a specific fixed-point type to be provided by context to use the result.]

4.5.6. Highest Precedence Operators

In accordance with AI-00868, exponentiation by an integer is equivalent to repeated multiplication, but with arbitrary association of operands (that is, not necessarily left to right), followed by a final reciprocal for the case of a real operand with a negative exponent.

[*Note*: Arbitrary association allows exponentiation by an integer to be performed using repeated squaring, rather than strictly as a sequence of single multiplications.]

[*Note*: Section 4.5.7, Accuracy of Operations with Real Operands, is moved to Annex NM:ALL]

4.6. Type Conversions

This section defines rules that govern whether it is legal to convert from one type to another. In addition, it specifies checks that are performed at run time as part of a type conversion. A type is *convertible* to a second type if and only if it is legal to convert from the first type to the second, according to the rules in this section.

In what follows, the *ancestor* of a specific type T or class-wide type T’CLASS is defined to be the type T itself, or any type from which T is derived, directly or indirectly.

A conversion denotes a view of its operand. The resulting view is of the target subtype. For parameter passing purposes, a conversion of a variable is a variable, a conversion of a constant is a constant, and a conversion of a value is a value. The tag of a class-wide object or value is preserved through a series of conversions.

Similar to Ada 83, a conversion between types is generally permitted if their root types share a common ancestor. However, there are additional restrictions for tagged types, and additional allowances for general access types.

The basic rules for conversions between tagged types are as follows:

- The target type must cover an ancestor of the operand type, or the operand type must cover the target type;
- If only the latter case is true, then at run-time, a check is made that the target type covers an ancestor of the specific type identified by the tag of the operand. CONSTRAINT_ERROR is raised if this check fails.

In addition to conversions between access types with a common ancestor (which are allowed in Ada 83), Ada 9X allows conversions to access-to-subprogram types and general access types:

- For conversion to an access-to-subprogram type, the operand's designated subprogram specification must be subtype conformant with that of the target type.
- It is illegal to convert from an access-to-constant type to an access-to-variable type.
- For conversion to a general access type, the operand and target designated types must be convertible. If there is a run time tag check associated with converting from the operand designated type to the target designated type, this check is performed if the operand is not null. CONSTRAINT_ERROR is raised if this check fails.
- For conversion to a general access type, the target designated subtype constraints must statically match those of the operand designated subtype, unless the target designated subtype is an unconstrained subtype with discriminants. The same representation clauses and pragmas must apply to both designated subtypes.
- For conversion to an anonymous access type (see 3.9), as part of a parameter association or discriminant constraint, a check is made that the access value is not null. CONSTRAINT_ERROR is raised if this check fails.
- For conversion of an access parameter (see 3.9) to a general access type, if the scope level of the access parameter is deeper than that of the target type, then a check is made at run time that the scope level of the designated object is no deeper than that of the target type. CONSTRAINT_ERROR is raised if this check fails. [*Implementation note*: This implies that the scope level must be passed in with an access parameter, in a manner similar to the CONSTRAINED attribute.]
- For any other access type conversion, the scope level (see 8.2) of the operand type must not be deeper than that of the target type.

[*Note*: Implicit type conversions are no longer involved with overload resolution. Instead, the formal parameter type is the expected type, and the actual type must match the expected type according to the operand matching rules (see 3.3.3). Any implicit conversion is applied after overload resolution (see 6.4.1).]

Implicit subtype conversions for arrays are generalized to apply in all circumstances where an applicable constraint is present (see RM 4.3.2(4-8)), except for **out** and **in out** actual parameters (see 6.4.1). However, an implicit subtype conversion is never applied to an array aggregate with an **others** choice (see 4.3.2).

4.7. Qualified Expressions

The expected type of the operand of a qualified expression is that of the type mark.

4.8. Allocators

[*Note*: See 3.9.2 for a discussion of allocators.]

4.9. Static Expressions and Static Subtypes

Static expressions involving only the operators of the root numeric types are evaluated exactly. [*Note*: This is equivalent to the Ada 83 rule on static universal expressions.]

The rules for static expressions and static subtypes are generalized to allow more kinds of compile-time-known expressions to be used where static values are required, as follows:

- A basic operation may appear in a static expression.
- The scalar attributes of statically constrained objects or subtypes are static.
- The discriminants of statically constrained objects are static.
- The RANGE attribute of a statically-constrained array subtype or object is a static range.
- A type conversion is static if the type mark denotes a static subtype and the operand is a static expression. No rounding is performed except on conversion from real to integer, in which case, if the value is exactly between two integers, the result is rounded away from zero.

4.9.1. Statically Matching Constraints and Subtypes [*new*]

A constraint *statically matches* another constraint if both are static and equal, or both are non-static and originate from the same subtype indication.

A subtype statically matches another subtype if both have the same base type, and both are unconstrained or have statically matching constraints.

A discrete constraint is *statically compatible* with a subtype if it statically matches the constraint of the subtype, or if both are static and the constraint is compatible with the subtype. In the case of a discriminant used to define an array bound, the constraint of the discriminant's subtype is also statically compatible with the index subtype if both subtypes are static and no value of the discriminant could cause a constraint error. [*Note*: This extra rule accommodates the possibility of a null array.]

[*Note*: Statically matching constraints and subtypes are the basis for parameter subtype conformance checking (see 6.3.1).]

[*Note*: The section on Universal Expressions is removed. It is subsumed by 4.5.5 and 4.9.]

5. Ada 9X Statements

5.1. Simple and Compound Statements - Sequences of Statements

5.2. Assignment Statement

For an assignment statement whose left-hand side is of a specific type, the expected type of the right-hand side is that same type. For an assignment statement whose left-hand side is of the class-wide or universal type for a class, the expected type of the right-hand side is the root type of the class.

5.3. If Statements

5.4. Case Statements

5.5. Loop Statements

5.6. Block Statements

5.7. Exit Statements

5.8. Return Statements

5.9. Goto Statements

6. Ada 9X Subprograms

6.1. Subprogram Declarations

The syntax for subprogram declarations is revised to allow ‘‘**is** <>’’, which means that the subprogram is *abstract*. The syntax for parameter specification is revised to allow for access parameters (see 3.9).

```
subprogram_declaration ::=
  subprogram_specification [is <>];

parameter_specification ::=
    identifier_list : mode type_mark [:= expression]
  | identifier_list : access_definition [:= expression]
```

An abstract subprogram must not have a body. Only a primitive operation of a tagged type may be declared abstract. An abstract subprogram may only be called with class-wide controlling operands (see 3.4.2).

If a primitive operation of a tagged type is abstract, then the type is an abstract type (see 3.4.2). When deriving from an abstract type, the abstract operations must be overridden, either with non-abstract declarations, or with another abstract subprogram declaration. [*Note*: Hence, a tagged type is only abstract if one of its primitive operations is explicitly declared as abstract.]

A formal parameter specified by an access definition is an **in** parameter of an anonymous access type (see 3.9). Such a parameter is called an *access parameter*.

6.2. Formal Parameter Modes

Whether an implementation must pass a parameter by copy is determined as follows:

- A parameter must be passed by copy if its type is elementary, or is a private type whose full type must be passed by copy.
- Otherwise, it is implementation dependent whether a parameter is passed by copy or by reference. [*Note*: In Ada 83, a program was erroneous if its effect depended on the parameter passing method chosen. In Ada 9X, such a program is non-portable.]

An **out** parameter is a variable that permits both reading and updating. If an **out** parameter is passed by copy, then any subcomponent that is a discriminant, is of an access type, or has a default initial expression, is copied in prior to the call. It is a bounded error to read a scalar **out** parameter, or any scalar subcomponent of an **out** parameter that is not required to be copied in, prior to initializing it.

6.3. Subprogram Bodies

A renaming declaration may act as a body for a subprogram. The renaming must be of a subprogram that is subtype conformant with the specification given in the renaming declaration (see 6.3.1).

[*Note*: In accordance with the April ’92 ISO WG9 meeting, we have dropped the proposal allowing a generic instantiation to act as a body. Renaming may be used to accomplish the same goal, if necessary.]

6.3.1. Conformance Rules

Each subprogram has a *calling convention*, which is a language name (see 13.9). The default calling convention of a subprogram with a normal Ada body is language ADA. This default may be overridden using pragmas IMPORT, EXPORT, or LANGUAGE (see 13.9). The calling convention for a predefined or *intrinsic* operation is language INTRINSIC. An intrinsic operation is an operation whose specification is defined in Ada, but whose implementation may be incorporated directly into the compiler (that is, there need be no explicit Ada body for the operation).

The subprogram specification conformance rules, as well as discriminant part conformance rules, are relaxed to require only the static semantic equivalence defined below as ‘‘full conformance.’’

There are four levels of conformance in Ada 9X:

Type Conformance requires that two specifications have the same number of parameters (and both have a result if either does) and at each parameter position corresponding parameters (and results, if present) have the same (base) type, or in the case of access parameters (see 3.9), the same designated type.

Mode Conformance
requires type conformance and that the parameters at each corresponding position have identical modes. For access parameters (see 3.9), the designated subtypes must statically match.

Subtype Conformance
requires mode conformance. In addition:

- The subtypes of the parameters at each parameter position (and results, if any) must statically match (see 4.9.1);
- The calling conventions must be the same;
- Neither calling convention can be INTRINSIC;
- If one is a protected subprogram or entry, then the other must be a protected subprogram or entry, respectively (see 9.7).

Full Conformance requires subtype conformance and that the formal parameters at corresponding positions have the same names and conforming default expressions (if any). Two default expressions are said to conform if each primary, operator, or basic operation within one expression has a corresponding primary, operator, or basic operation within the other expression, and, in the case of a primary that is a name, they (statically) denote the same entity.

6.3.2. Inline Expansion of Subprograms

6.4. Subprogram Calls

6.4.1. Parameter Associations

The rules for parameter associations are as follows:

- The type of the actual parameter must match the formal parameter, according to the operand matching rules (see 3.3.3);
- The type of the actual parameter must be convertible to the type of the formal parameter (see

4.6);

- For an **in** or **in out** parameter that must be passed by copy (see 6.2) the value of the actual parameter is converted (see 4.6) to the subtype of the formal parameter and assigned to it, which may raise CONSTRAINT_ERROR.
- After a call that does not propagate an exception, for an **in out** or **out** parameter that must be passed by copy, the value of the formal parameter is converted to the subtype of the actual parameter and assigned to it, which may raise CONSTRAINT_ERROR.
- For an **in** parameter that may be passed by reference, the formal parameter is defined to be a (read-only) conversion (see 4.6) of the actual parameter, to the subtype of the formal parameter. This conversion may raise CONSTRAINT_ERROR.
- For an **in out** or **out** parameter that may be passed by reference, the value of the actual parameter is checked to ensure that it satisfies the constraints of the subtype of the formal parameter. CONSTRAINT_ERROR is raised if this check fails. The formal parameter is defined to be an (updatable) view of the actual parameter. [*Note*: As in Ada 83, no constraint check is performed on return for a parameter that may be passed by reference.]

As in Ada 83, for a parameter that may be passed by reference, it is implementation dependent whether the value of the formal parameter (which is a view of the actual) is held in a local copy between the call and the return. If an exception occurs, it is therefore implementation dependent whether the actual is updated for such a parameter, when its mode is **in out** or **out**.

[*Note*: The rules for parameter associations apply to discriminant constraints as well, where all discriminants are handled like formal **in** parameters that must be passed by copy. See 3.6.1 and 3.6.2.]

6.4.2. Default Parameters

6.5. Function Subprograms

6.6. Parameter and Result Type Profile - Overloading of Subprograms

6.7. Overloading of Operators

The equality operator "=" may be overloaded for any combination of parameter and result types. A corresponding inequality operator "/=" is implicitly declared only if the result type is STANDARD.BOOLEAN. An inequality operator "/=" may be explicitly declared if its result type is not STANDARD.BOOLEAN.

7. Ada 9X Packages

7.1. Package Structure

7.2. Package Specifications and Declarations

7.3. Package Bodies

7.4. Private Type and Deferred Constant Declarations

7.4.1. Private Types

The syntax for a private type declaration is augmented to allow the specification of a tagged private type.

```
private_type_declaration ::=
  type identifier [discriminant_part]
    is [tagged] [limited]  private;
```

A private type is a composite type (see 3.3) outside of the scope of the full type declaration. (In this section a task or protected type declaration is considered a ‘‘full’’ type declaration.)

If a private type declaration has a discriminant part, then the full type declaration must be a composite or derived type declaration with a fully conforming discriminant part (see 6.3.1).

If a private type declaration includes the modifier **tagged**, then the full type declaration must be tagged.

7.4.2. Private Extension [*new*]

The declaration of a private type extension of a tagged parent type is allowed in the visible part of a package. A private extension is a tagged composite type, with a private extension part.

For each private extension declared in the visible part of a package, a corresponding full type declaration must appear in the private part for a type derived (directly or indirectly) from the tagged parent type. [*Note*: This allows the full type to be defined by deriving from a type produced by a generic instantiation, or one or more intermediate derivations.]

Unless explicitly overridden within the visible part, the implementations of the primitive operations inherited from the parent type of a private extension are determined by the full type declaration and the other declarations in the private part. Unless a primitive operation of a tagged private type or private extension is declared as abstract in the visible part, the full type must not be an abstract type. [*Note*: These rules imply that although the specifications of the primitive operations of a private extension come from the parent type identified in the private extension definition, the implementations come from the full type. They also imply that a type is abstract only if some visible operation is explicitly declared as abstract.]

Example of a private extension:

```
with BASIC_WINDOWS;   -- defines tagged type ROOT_WINDOW
package EXTENDED_WINDOWS is
    type LABELED_WINDOW(LENGTH : NATURAL) is
      new BASIC_WINDOWS.ROOT_WINDOW with private;
    . . .
private
    type LABELED_WINDOW(LENGTH : NATURAL) is
      new BASIC_WINDOWS.ROOT_WINDOW with
        record
          LABEL : STRING(1..LENGTH);
          FONT : FONT_ID := 0;
        end record;
end EXTENDED_WINDOWS;
```

7.4.3. Operations of a Private Type [*originally 7.4.2*]

[*Note*: As in Ada 83, the attribute T'CONSTRAINED of a private subtype T being true is not the same as T being ‘‘constrained.’’ In particular, if the full subtype is an elementary type, T'CONSTRAINED is always true, even if no constraints apply to the subtype. See RM 3.3(4) and RM 7.4.2(10).]

7.4.4. Deferred Constants [*originally 7.4.3*]

A deferred constant declaration, for a constant of any type, is allowed immediately within the visible part of a package. The full constant declaration must appear immediately within the private part of the same package. [*Note*: In Ada 83, deferred constants were restricted to being of a private type that is declared in the same visible part (see RM 7.4(4)). We are relaxing this restriction.]

7.4.5. Limited Types [*originally 7.4.4*]

[*Note*: We are still studying the issue of returning a value of a limited type. This section is our initial attempt to define the semantics of such an operation, as well as other issues relating to limited types. Fundamentally, many of these issues revolve around the question: ‘‘what is the *value* of an object of a limited type?’’ (We will grinningly ignore opinions that "objects of a limited type have no value.")]

The value of an inherently aliased (see 3.9.2) object of a limited type *designates* that object. [*Note*: This rule is an adaptation of the Ada 83 rule for task types, and at least opens the possibility for lessening the current strong distinction between task objects and tasks.] If a function returns a value of an inherently aliased limited type, the result designates the same object designated by the value of the return expression. If the object is local to the function, it is moved if necessary, so that the storage for the (moved) object is not reclaimed and the object is not finalized (see 7.4.6) prior to the caller finishing with the returned value.

It is a bounded error to return the value of an inherently aliased local object if the object cannot be moved. Task objects and objects containing access discriminants cannot be moved. As a result of the bounded error, PROGRAM_ERROR may be raised, or a value may be returned that designates a temporary object of the same type that is local to the caller, which is reclaimed when the caller is finished with the returned value. [*Note*: This addresses the issue of returning a task outside of the scope of its master.]

[*Note*: It is interesting to note that in Ada 83, an object containing a task subcomponent is considered implicitly referenced as long as the task is not terminated (see RM 4.8(7)), implying that the task might have access to the enclosing object, and that the object is inherently aliased. However, this might appear inconsistent with the note in RM 13.10.1(8) saying that UNCHECKED_DEALLOCATION of a task object has no effect on the associated task. Objects containing protected subcomponents will have similar considerations, particularly when the protected subcomponent is used to provide a relatively long-lived lock on the enclosing object. We will be studying these issues further during the revision.]

7.4.6. Controlled Types [*new*]

A limited tagged type CONTROLLED is defined in package SYSTEM. Any type derived from this type, directly or indirectly, is called a *controlled* type. The tagged type CONTROLLED has two primitive, dispatching operations, INITIALIZE and FINALIZE, with specifications as follows:

```
    procedure INITIALIZE(OBJECT :  in out CONTROLLED);

    procedure FINALIZE(OBJECT :  in out CONTROLLED);
```

These operations have null implementations for the type CONTROLLED. They may be overridden with new implementations for any type derived from CONTROLLED.

After a controlled object (including a subcomponent of another object) is created and has been default initialized, the operation INITIALIZE is applied to it as the last step in its initialization. If a controlled object has controlled subcomponents, the INITIALIZE operation is applied to the controlled subcomponents prior to applying it to the enclosing object. [*Note*: This allows the INITIALIZE operation for a composite object to refer to its subcomponents knowing they have been properly initialized.] For an object created by an allocator, any subcomponent task activations follow all INITIALIZE operations.

Prior to reclaiming the storage for a controlled object (including a subcomponent of another object), the operation FINALIZE is applied to it. [*Note*: This also applies to objects whose storage is reclaimed by UNCHECKED_DEALLOCATION or automatic storage reclamation.] If a controlled object has controlled subcomponents, the FINALIZE operation is applied to the enclosing object prior to applying it to its controlled subcomponents. [*Note*: This allows the FINALIZE operation for a composite object to refer to its subcomponents prior to their being finalized.]

Upon leaving a declare block, or a procedure, entry, or task body, the FINALIZE operation is applied to all local controlled objects, including those in local access collections that have not already been finalized by UNCHECKED_DEALLOCATION or automatic storage reclamation. For declared objects and their subcomponents, the FINALIZE operations are applied in the reverse of the order of the INITIALIZE operations. The FINALIZE operations are applied after waiting for any local tasks to terminate. [*Note*: Like task waiting, the FINALIZE operations are performed whether the scope is left by reaching the last statement, a return, an exit, a goto, or an abort or asynchronous transfer.]

In the case of a function that returns the value of a local controlled object or local object with a controlled subcomponent (see 7.4.5), the FINALIZE operation(s) on this object (and possibly others in the same scope) is (are) deferred until the caller of the function is finished with the returned value. For all other functions, the FINALIZE operations are applied upon leaving the function body, as with a procedure.

Immediately prior to program termination, FINALIZE operations are applied to library-level controlled objects (including those in library-level access collections, except those already finalized). This occurs after waiting for library-level tasks to terminate. [*Note*: We are considering a pragma that would apply to a controlled type that would suppress FINALIZE operations for library-level objects of the type upon program termination. This would be useful for types whose finalization actions consist of simply reclaiming global heap storage, when this is already provided automatically by the environment upon program termination.]

If any FINALIZE operation propagates an exception, then after completing any other FINALIZE operations due to be performed, PROGRAM_ERROR is raised at the point where normal execution would have continued. For example, upon leaving a block due to a goto statement, the PROGRAM_ERROR would be raised at the point of the target statement named by the label.

While performing an INITIALIZE or FINALIZE operation of a controlled type, abort and asynchronous transfer of control are deferred. [*Note*: This ensures that an object is never partially initialized when finalized, or partially finalized when reclaimed. This also allows, for example, the result of an allocator performed in an INITIALIZE operation to be stored in an access object without being interrupted in the middle of the assignment statement, ensuring that the associated storage can be fully reclaimed by the FINALIZE operation.]

[*Note*: Because CONTROLLED is a library-level type, all controlled types will also be library-level types (see 3.4.1). This ensures that the FINALIZE operations may be applied without providing any ‘‘display’’ or ‘‘static-link.’’ This simplifies finalization as a result of automatic storage reclamation, abort, and asynchronous transfer of control.]

7.5. Example of a Table Management Package

7.6. Example of a Text Handling Package

8. Ada 9X Visibility Rules

8.1. Declarative Region

A protected record unit is a declarative region.

8.2. Scope of Declarations

Every declaration, every entity declared by a declaration and every object has a *scope level*, defined as follows:

[*Note*: We intend to simplify this discussion as part of the revision process.]

All declarations that are local to (i.e. declared immediately within) a particular declarative region have *the same* scope level. The scope level of a declaration that is local to a package is *the same as* that of the package. The scope level of a declaration that is local to a subprogram, task unit, protected unit, or block statement is *deeper than* that of the program unit or block statement. The scope level of a declaration that is local to a generic declaration is defined as for the instance. The scope level of a declaration that is local to a generic body is *deeper than* that of the generic declaration. The scope level of package STANDARD is called the *library-level scope*, and entities with this scope level are called *library-level* entities.

The scope level of a derived access type is that of the parent type. The scope level of an access parameter type is that of the corresponding access discriminant or parameter (see 3.6.1 and 3.9). The scope level of any other entity declared by a declaration is that of the declaration.

The scope level of an object designated by an access parameter (see 3.9) is that of the object designated by the actual expression initializing the access parameter. The scope level of an object designated by an access discriminant (see 3.6.1) is that of the discriminated object. The scope level of an object designated by any other access value is that of the access type. The scope level of a subcomponent of an object is that of the object. The scope level of a protected operation of an object is that of the object.

On a subprogram call, the scope level of a parameter or result of the subprogram being called is deeper than any object declared by or accessible to the caller prior to the call. [*Note*: This definition ensures that there is no scope check as part of initializing an access parameter (see 3.9 and 4.6). It also ensures that an access discriminant of the result of a function call cannot be used to initialize an access discriminant of an object declared by the caller (see 3.6.1 and 4.6).]

[*Notes*:

A package does not introduce a deeper scope level. Other program units do.

In order to prevent dangling pointers of various kinds, we have introduced certain *scope checks*, based on the above definition of scope levels. There are scope checks on derived tagged type declarations (see 3.4.1), on the ACCESS attribute (see 3.9.2), and on type conversion of access types (see 4.6). Most scope checks are done at compile time; the only exception is for conversion of an access parameter (see 3.9) to a general access type (see 4.6).]

8.3. Visibility

The declaration of a protected operation of a given protected record type is visible by selection at the place of the selector after the dot of a selected component whose prefix is appropriate for the protected record type. (See RM 8.3(7-12).)

For overloadable entities, the definition of *homograph* is revised as follows:

Two declarations of overloadable entities are homographs if they have the same identifier, operator symbol, or character literal, the same number of parameters and results, and if, according to the operand matching rules (see 3.3.3):

- the result types (if any) could both match the same specific type; and
- for each parameter position, a single specific type could match both formal parameters.

[*Note*: This rule makes two declarations homographs if they are unresolvable even in a context that determines all specific operand and result types.]

8.4. Use Clauses

For a child library unit (see 10.1) referenced in a with clause, the declaration of its simple name, which logically occurs within the child part of its parent, is made directly visible by a use clause for the parent package.

[*Note*: In response to the April '92 ISO WG9 meeting, we are proposing two new kinds of use clauses, to address the visibility of primitive operators and the accessibility of private declarations of an ancestor library unit.] For Ada 9X, there are two new kinds of use clauses:

```
use_clause ::=
    ... -- as in Ada 83
  | use type type_mark {, type_mark};
  | use private ancestor_library_package_name
       {, ancestor_library_package_name};
```

A *type-based* use clause is one introduced by the reserved words **use type**. A *primitive operator* of a type is a primitive operation of a type whose designator is an operator symbol. A type-based use clause makes directly visible each primitive operator of the type(s) determined by the type_mark(s), except within the immediate scope of a homograph of the primitive operator.

A use clause introduced by the reserved words **use private** makes the private declarations of an ancestor library unit package directly accessible within a child library unit (see 10.1). Outside the scope of such a use-private clause, it is illegal in a child library unit to reference a declaration appearing within the private part of an ancestor library unit package. A use-private clause has no affect on overload resolution; the check for reference to private declarations is performed after overload resolution. [*Note*: This approach is designed to avoid ‘‘Beaujolais’’ effects.]

8.5. Renaming Declarations

[*Note*: A renaming declaration may act as a body for a subprogram. See 6.3.]

8.6. The Package STANDARD

8.7. The Context of Overload Resolution

Rather than a preference rule against implicit conversion, as defined in RM 4.6(15), in Ada 9X there is a preference for the primitive operators of the root numeric types *root_integer* and *root_real*. The rule is as follows:

- In a given context, if two legal interpretations differ only in that one is for a primitive operator of the type *root_integer* or *root_real*, and the other is for a corresponding primitive operator for another numeric type, the interpretation using the primitive operator of the root numeric type is *preferred*.
- For an innermost complete context (see RM 8.7(3)), if there is exactly one overall legal interpretation where each constituent's interpretation is the same as or preferred (in the above sense) over those in all other overall legal interpretations, then that one overall legal interpretation is chosen.

[*Note*: It is the intent that this preference rule is more locally enforceable than that of RM 4.6(15). It should also minimize interpretation shifts due to the addition or removal of a use clause (the so called *Beaujolais* effect).]

9. Ada 9X Tasks and Synchronization

9.1. Task Specifications and Task Bodies

A discriminant part is allowed in a task type specification. The discriminants may be referenced within the specification and body of the task using the simple name of the discriminant. In a task type specification, the simple name of a discriminant denotes the discriminant of the task object being elaborated. In a task body, the simple name of a discriminant denotes the discriminant of the task object designating the task currently executing the body.

[*Note*: As with any discriminated type, the discriminants may also be referenced as a selected component of a task object (see RM 4.1.3(4,5)).]

```
task_specification ::=
  task [type] identifier [discriminant_part] [is
    {entry_declaration}
    {representation_clause}
 [  private
    {entry_declaration}
    {representation_clause}]
  end [task_simple_name]]
```

The first list of entry declarations of a task specification is called the *visible part* of the task specification. The optional list of entry declarations after the reserved word **private** is called the *private part* of the task specification. An entry declared in the private part is not visible outside the task unit.

[*Note*: The task private part has been retained pending further study and review, to be consistent with protected types and requeue.]

9.2. Task Types and Task Objects

There is no limitation on the use of an **out** parameter of a task type, since there are no remaining limitations associated with **out** parameters of a limited type (see 6.2). As in Ada 83, for all parameter modes, if an actual parameter designates a task, the associated formal parameter designates the same task; the same holds for a subcomponent of a parameter (RM 9.2(2)).

Expressions appearing within representation clauses and pragmas in a task specification are reevaluated for each object of the task type.

Analogous to tagged types (see 3.4.1) a class-wide type SYSTEM.TASK_CLASS is defined for the task class. By definition, all task types are covered by SYSTEM.TASK_CLASS. SYSTEM.TASK_CLASS is used for the explicit definition of class-wide operations on tasks, such as dynamic priority setting operations (see the Real-Time Systems Annex).

9.3. Task Execution - Task Activation

9.4. Task Dependence - Termination of Tasks

9.5. Protected Specifications and Protected Bodies [*new*]

[*Note*: We have adopted the terms ‘‘protected type’’ and ‘‘protected object’’ rather than protected record type or object, since protected types are not considered record types.]

A protected unit consists of a protected specification and a protected body. A protected specification that starts with the reserved words **protected type** and an identifier declares a protected type and an unconstrained subtype. The identifier is the name of the subtype.

A protected specification without the reserved word **type** defines a *single protected object*. A protected specification with this form of specification is equivalent to the declaration of an anonymous protected type immediately followed by the declaration of an object of the protected type. The protected unit identifier names the object. In the remainder of this chapter, explanations are given in terms of protected type declarations. The corresponding explanations for single protected object declarations follow from the stated equivalence.

The value of an object of a protected type designates the protected object, having protected operations and protected components. Each protected specification must have a corresponding protected body. Each protected operation must have a corresponding body; the rules are similar to those for Ada 83 subprogram bodies. The execution of the protected operations are specified by the protected operation bodies given in the body of the protected unit.

```
protected_declaration ::= protected_specification;

protected_specification ::=
  protected [type] identifier [discriminant_part]  is
    { protected_operation_declaration }
    private
    { protected_operation_declaration }
    record
      component_list
    end [protected_simple_name]

protected_operation_declaration ::=
    subprogram_declaration
  | entry_declaration

protected_body ::=
  protected body protected_simple_name  is
    { protected_operation_item }
  end [protected_simple_name];

protected_operation_item ::= subprogram_declaration
  | subprogram_body
  | entry_body
```

9.6. Protected Types and Protected Objects [*new*]

A protected type is a limited composite type. A protected object is an object whose type is a protected type.

For all parameter modes, if an actual parameter is a protected object, the associated formal parameter denotes the same protected object; the same holds for a subcomponent of a parameter.

The first list of protected operation declarations of a protected specification is called the *visible part* of the protected specification. The declarations and component list that appear after the reserved word **private** make up the *private part* of the protected specification.

The discriminants and the protected operations declared in the visible part of the protected specification are visible by selection outside the protected unit. The non-discriminant data components of a protected object are only visible within the body of the protected unit. They refer to components of the protected object passed as an implicit parameter (see 9.7 below).

Expressions appearing within pragmas and default expressions in a protected specification are reevaluated for each object of the protected type.

[*Note*: As for other discriminated types, the discriminants of a protected object may be named with selected component notation anywhere the object may be named.]

Example of a Protected Type:

```
    protected type COUNTING_SEMAPHORE(INITIAL_COUNT : INTEGER := 1) is
      -- A Counting Semaphore,
      --   Acquire = ''P'' operation
      --   Release = ''V'' operation
        function  COUNT return INTEGER;  -- Current count
        procedure RELEASE;     -- Release a resource
        entry     ACQUIRE;     -- Acquire a resource, suspend if none
    private record
        CURRENT_COUNT : INTEGER := INITIAL_COUNT;
          -- Count of available resources, default initial value is 1;
          -- May be negative to represent being in ''debt.''
    end COUNTING_SEMAPHORE;

    protected body COUNTING_SEMAPHORE is
        function COUNT return INTEGER is
          -- Return current count of available resources
        begin
            return CURRENT_COUNT;
        end COUNT;

        procedure RELEASE is
          -- Release a resource for acquisition by another caller
        begin
            -- Increment count of available resources
            CURRENT_COUNT := CURRENT_COUNT + 1;
        end RELEASE;

        entry ACQUIRE when CURRENT_COUNT > 0 is
          -- Acquire a resource when some are available;
          -- Suspend if none are available.
        begin
          -- Decrement count of available resources
            CURRENT_COUNT := CURRENT_COUNT - 1;
        end ACQUIRE;
    end COUNTING_SEMAPHORE;
```

9.7. Protected Operations, Entry Calls, and Accept Statements [*originally 9.5*]

Protected objects may have *protected operations* that are functions, procedures, or entries. [*Note*: As in Ada 83, a task may only have entries.]

Within a protected or task unit, the name of a protected operation is the protected operation's simple name. Outside a protected or task unit, the name of a protected operation declared in the visible part has the form of a selected component. The prefix must be appropriate for (see RM 4.1(6)) the protected or task type that declares the operation.

The body of a protected unit may contain function, procedure, and entry bodies corresponding to the operations declared in the protected specification. In addition, the body of a protected unit may contain declarations and bodies for local subprograms. These are not visible outside the protected unit.

```
entry_body ::=
  entry identifier
   [(for identifier  in discrete_range)] [formal_part]
    when condition
 [is
   [declarative_part]
  begin
    sequence_of_statements
 [exception_handler_part]
  end [entry_simple_name]];
```

An entry body specifies a *barrier condition* for an entry after the reserved word **when**, and optionally specifies the execution of an operation to be associated with the entry. If the entry is a family, then the name of the family index is provided using the *iterator* syntax **for** identifier **in** discrete_range.

The barrier condition is an expression of a boolean type; when the barrier evaluates to TRUE, the entry body may be executed. The barrier may not reference any formal parameters to the entry, but it may reference anything else visible, including the entry family index, the components (including discriminants) of the protected object, the COUNT attribute of an entry of the protected object, or data global to the protected unit.

[*Note*: The restriction against referencing the formal parameters within a barrier expression ensures that all calls in the same entry queue see the same barrier value.]

A protected operation takes a protected object as an implicit parameter. This parameter has mode **in** for protected functions and mode **in out** for protected procedures and entries.

A *protected action* is a synchronized operation on a protected object. A call on a protected operation from outside the protected unit is a protected action. Prior to beginning the protected action, the operation name and the actual parameters are evaluated. A protected action is synchronized with other protected actions on the same protected object, as follows:

- No two protected actions on the same protected object may proceed concurrently, unless both are the result of function calls.
- As the first step for an action that results from an entry call, the barrier condition is evaluated. If it yields FALSE, the call is queued, and does not proceed further until the barrier is reevaluated and no longer yields FALSE.
- As the final step of a protected action that is not the result of a function call, if any calls are

queued, the barrier conditions are reevaluated. If the barrier of an entry with a queued call no longer evaluates to FALSE, one call on such a queue is selected as the next protected action to be performed on the protected object. If more than one call is eligible, an implementation-defined rule is used to select among them.

The process of reevaluating the barrier conditions to select the next action is called *servicing the entry queues*. By default, the calls on an entry queue are serviced in order of arrival. Other orders may be specified by a language-defined or implementation-defined pragma (see RT:QUEUE-POLICY:SECT). The language does not specify which task services the entry queues, and executes the entry body of the selected action.

If the evaluation of a barrier condition propagates an exception, PROGRAM_ERROR is propagated to all callers currently on any entry queue of the protected object.

An implementation need not reevaluate a barrier condition if its associated queue is empty, or if the evaluation of a prior barrier allows a call to be selected. A barrier condition also need not be reevaluated if the variables referenced by it were not altered by a protected action on the protected object since the last evaluation of the barrier. [*Note*: These rules allow certain simplifications and optimizations in the implementation. The second rule implies that barrier conditions should not reference variables altered outside of the protected unit.]

The value of the COUNT attribute of an entry of a protected object is the number of calls in its entry queue. The COUNT attribute is only available within the body of the protected unit.

When an entry call is cancelled (see 9.9.2), the value of the COUNT attribute will still include this call until either it is removed from the queue, or the call is selected. If a cancelled call is selected before it is removed from the queue, the call will be ignored.

Removing a cancelled call from its queue is a protected action, and hence cannot occur during any other protected action. Upon completing the removal, the entry queues whose barriers depend on the COUNT attribute are serviced.

A protected action may include a call on a protected subprogram where the target object is identified implicitly to be the current protected object. In this case, a new protected action is not started; instead, the call is performed as part of the current action. [*Note*: the parameter mode of the implicit protected object parameter precludes a protected function from calling a protected procedure in this way.]

A protected action may include a call on a protected subprogram where the protected object is specified as the prefix. In this case, a new protected action is performed on the specified protected object, as described above.

The following are defined to be *potentially suspending* operations:

- a select statement;
- an accept statement;
- an entry call;
- a delay statement;
- a task creation;

- an abort statement;
- a call on a subprogram in a language-defined I/O package;
- a call on an implementation-defined subprogram that is defined to potentially suspend the calling task.

It is a bounded error for a protected action to directly or indirectly perform a *potentially suspending* operation. It is also a bounded error for a protected action on a protected object to call, directly or indirectly, a protected subprogram with a prefix identifying an object that is the same protected object.

In both cases, the bounded error may eventually result in deadlock, raise PROGRAM_ERROR (immediately or as soon as the error is detected), or go undetected. To ensure the error is bounded, an implementation that does not detect all such errors, and allows a second protected action to be improperly performed in the middle of the current protected action, must assume that the state of the protected object is updated by any operation that might lead to such an undetected error (such as a call out from the protected type).

[*Note*: Some of the above ‘‘potentially suspending’’ operations are not directly suspending. However, they are still treated as errors because requiring them to be supported from within a protected operation was felt to impose an undesirable implementation burden.]

9.7.1. Requeue Statements [*new*]

A requeue statement may be used to complete the execution of an entry body or accept statement, and at the same time commence a new entry call. In effect, the original entry call is being redirected to a new entry.

requeue_statement ::= **requeue** *entry*_name [**with abort**];

A requeue statement is only allowed within an entry body or an accept statement, and is associated with the innermost such construct; a requeue statement is not allowed within a program unit body nested within this entry body or accept statement. A requeue within an entry body may only be to an entry of a protected object. A requeue within an accept statement may be to an entry of a task or protected object.

The entry named by the requeue statement must either have no parameters, or be subtype conformant (see 6.3.1) with the parameter profile of the innermost enclosing entry body or accept statement. [*Note*: The presence or absence of a family index plays no part in the conformance check. The family index, if any, is considered part of the entry name, not one of its parameters.]

The execution of a requeue statement proceeds by first evaluating the entry name, including any prefix identifying the target task or protected object. If the target object is identified implicitly as being the current task or protected object, the call is simply added to that entry’s queue, to be selected later when the entry is serviced.

If the target object is specified explicitly in the entry name, then the requeue is handled like a normal entry call up until the point when the requeued call is complete or queued.

Having queued or completed the requeued call, the entry body or accept statement containing the requeue statement is exited.

A requeued entry’s formal parameters are handled as follows:

- If the new entry has formal parameters, then the values of these formal parameters will be initialized from the values of the corresponding formal parameters of the current entry at the time of the requeue.
- If the new entry does not have formal parameters, then the values of the formal parameters of mode **in out** or **out** of the current entry that were passed by copy are retained for later copy back and constraint check after completing the new entry call.

[*Note*: This treatment of formal parameters as part of a requeue will normally not require any special action at run-time, since the parameters to an entry call will normally be kept in a parameter block allocated by the original caller.]

If the requeue statement includes the reserved words **with abort** then abort is not deferred while the task is waiting on the new entry queue. Otherwise, abort remains deferred during the requeuing. See 9.12 for a further discussion of abort.

9.8. Delay Statements, Duration, and Time [*originally 9.6*]

The syntax for the **delay** statement is extended as follows:

delay_statement ::= **delay until** simple_expression;
  | **delay** simple_expression;

The first form of the delay statement delays the task until the specified time is reached. The type of the time expression must be either CALENDAR.TIME, or else some time type as allowed by Annex RT:ALL. If the time has already passed, the task is not suspended.

The second form of the delay statement delays the task for the specified duration. The type of the duration expression must be STANDARD.DURATION. The task is not suspended if the duration is less than or equal to zero.

[*Note*: Even if the task is not suspended, a delay statement is always considered a synchronization point for abort (see RM 9.10(6)).]

9.9. Select Statements [*originally 9.7*]

There are four forms of select statements. The first three are as in Ada 83. The last one provides asynchronous transfer of control, triggered by the completion of an entry call or the expiration of a delay (see 9.9.4).

select_statement ::= selective_wait
  | conditional_entry_call | timed_entry_call
  | asynchronous_select

9.9.1. Selective Wait [*originally 9.7.1*]

[*Note*: Having dropped multi-way entry call and iterators for select alternatives, the selective wait statement is back to its Ada 83 form (and name).]

9.9.2. Conditional Entry Calls [*originally 9.7.2*]

9.9.3. Timed Entry Calls [*originally 9.7.3*]

9.9.4. Asynchronous Transfer of Control [*new*]

An asynchronous select statement provides asynchronous transfer of control upon completion of an entry call or the expiration of a delay.

```
asynchronous_select ::=
  select
    triggering_statement [sequence_of_statements]
  then abort
    abortable_part
  end select;

triggering_statement ::= entry_call | delay_statement

abortable_part ::= sequence_of_statements
```

A task performing a delay statement is *queued abortably* until the delay expires. A task waiting on an entry queue is *queued abortably*, unless requeued without allowing abort.

[*Note*: The detailed semantics given below are still under study. The general model proposed is that the sequence of statements following the triggering statement is executed like a ‘‘software interrupt handler’’ prior to completing the finalization actions associated with the abortable part, and prior to performing the asynchronous transfer of control to the end of the select statement.]

For the execution of an asynchronous select, the triggering statement is attempted. If the statement completes without the task being queued abortably, then the sequence of statements following the triggering statement is executed.

If the task is queued abortably prior to completion of the triggering statement, then the abortable part begins execution. Thereafter:

- If the abortable part completes prior to the completion of the triggering statement, the triggering statement is aborted. If the triggering statement nevertheless completes, the sequence of statements following the triggering statement is executed. If the triggering statement is aborted prior to completion, the sequence of statements following the triggering statement is not executed.
- If the triggering statement completes prior to the completion of the abortable part, then the sequence of statements of the abortable part is aborted, and the sequence of statements following the triggering statement is executed. Any finalization actions associated with the abortable part are postponed until after executing the sequence of statements following the triggering statement.

See 9.12 for a discussion of the effect of aborting a sequence of statements.

Example of a main command loop for a command interpreter:

```
loop
    select
        TERMINAL.WAIT_FOR_INTERRUPT;
        PUT_LINE("Interrupted");
    then abort
        -- This will be abandoned upon terminal interrupt
        PUT_LINE("-> ");
        GET_LINE(COMMAND, LAST);
        PROCESS_COMMAND(COMMAND(1..LAST));
    end select;
end loop;
```

Example of a time-limited calculation:

```
select
   delay 5.0;
   PUT_LINE("Calculation does not converge");
then abort
   -- This calculation should finish in 5.0 seconds;
   --  if not, it is assumed to diverge.
   HORRIBLY_COMPLICATED_RECURSIVE_FUNCTION(X, Y);
end select;
```

9.10. Priorities [*originally 9.8*]

All discussion of priorities is deferred to Annex RT:ALL.

9.11. Task and Entry Attributes [*originally 9.9*]

9.12. Abort of a Task; Abort of a Sequence of Statements [*originally 9.10*]

The **abort** statement allows an entire task to be aborted. This renders the task uncallable (see RM 9.10(7)). The asynchronous select statement (9.9.4) allows a sequence of statements to be aborted.

Any task dependent on a task being aborted, or dependent on a master within a sequence of statements being aborted, is itself aborted. TASKING_ERROR is raised if a subsequent attempt is made to create a new task dependent on an aborted task or master.

Abort is deferred while a task is performing a protected subprogram or entry call, except while the task is queued on an entry queue with abort allowed.

Once abort is no longer deferred, the abort propagates out of the frame in which the task is executing. If the task as a whole is aborted, the abort propagates until the task completes. If a sequence of statements is aborted, the abort propagates until the sequence of statements is completed.

[*Note*: Annex RT:ALL specifies that abort must be *immediate*, once no longer deferred, for implementations conforming to the Real-Time Systems Annex.]

9.13. Shared variables [*originally 9.11*]

As in Ada 83, two tasks may communicate through variables visible to both tasks. However, it is erroneous to perform an update operation on a shared variable when any other operation on the shared variable might be performed. To ensure proper ordering between operations on shared variables, tasks must synchronize (for example, through the use of entry calls).

In Annex SP:ALL, additional pragmas are provided for identifying variables that require atomic or direct reading and writing.

[*Note*: A more formal memory model will be provided at a later date.]

9.14. Example of Tasking and Synchronization [*originally 9.12*]

10. Ada 9X Program Structure and Compilation Issues

10.1. Compilation Units - Library Units

Each library unit package (that is not an instantiation) must have a separate body. [*Note*: This eliminates several anomalies with the current handling of implicit trivial package bodies. This is not strictly upward-compatible. The work-around is to add a null package body to the end of a compilation containing a library package that does not need a body.]

A library unit is either a *root* library unit or a *child* library unit. A child library unit must have a library unit package as its *parent* library unit. The name of a child library unit is in the form of an expanded name (RM 4.1.3), with the prefix being the name of the parent library unit, and the selector being the identifier for the child library unit.

The syntax for subprogram_declaration, package_declaration, generic_declaration, generic_instantiation, subprogram_body, and package_body will be revised to allow expanded name syntax for the name introduced by the declaration:

program_unit_name ::= [*parent_library_unit_package*_name.]identifier

An expanded name is allowed in a program unit definition only if the unit is a library unit.

A child library unit is either a *visible* child or a *private* child of its parent package. A private child is not visible outside the tree of library units rooted at its parent library unit.

The syntax for compilation_unit and library_unit is revised as follows:

```
compilation_unit ::=
   context_clause [private] library_unit
  | context_clause secondary_unit
```

A private library unit is marked using the reserved word **private**.

[*Note*: The proposal for library unit renaming has been deleted in accordance with the April ’92 ISO WG9 meeting.]

In addition to the *visible part* and *private part* of a package, a library unit package also has an implicit *child part* where the declaration of a child unit is logically considered to occur when the child unit itself is compiled, or when the child unit is referenced in a with clause. The child part implicitly occurs at the end of the package specification, following the visible part and the private part (if any).

[*Note*: The role played by a parent package toward its child library units is analogous to the role played by package STANDARD to the root library units. In other words, root library units may be considered child units of package STANDARD. The specification of a parent package has no compilation dependence on its child units. The body of a parent package depends on a child unit only if the child is mentioned in a with clause applicable to the body.]

In the visible part of a visible child, the private declarations of the parent are not visible (neither directly nor by selection). However in the visible part of a private child, and in the private part and body of any child package, such declarations are directly visible. However, it is illegal to reference such declarations except within the scope of a use-private clause (see 8.4) that specifies the parent library unit’s name.

Note: The model whereby a child appears in the child part of its parent when compiled or referenced in a with clause has the following ramifications:

- The restrictions on ‘‘early’’ use of a private type (RM 7.4.1(4)) or a deferred constant (RM 7.4.3(2)) do not apply to uses in child units, because they follow the full declaration.
- Visible child subprograms are not derivable subprograms, because they are not declared within the same list of declarations as the type.
- When the parent package is ‘‘used’’ the simple names of the ‘‘with’’ed child units are directly visible (see 8.4).
- When a parent body ‘‘with’’s its own child, the simple name of the child is directly visible, and the parent package body must not include a declaration of a homograph of the child unit immediately within the declarative part of the body (RM 8.3(17)).

End of note.

Within a single program the expanded name of each program unit that is a compilation unit must be unique. Uniqueness of the expanded name must be enforced prior to execution of the program.

Example of library unit package and child library units:

```
package OS is
    -- Parent package defines types used throughout OS
    type FILE_DESCRIPTOR is private;
private
    type FILE_DESCRIPTOR is new INTEGER;
end OS;

procedure OS.INTERPRET(COMMAND : STRING);
    -- Pass a single string to the OS command interpreter

package OS.EXCEPTIONS is
    -- Define exceptions used throughout OS

    FILE_DESCRIPTOR_ERROR,
    FILE_NAME_ERROR,
    PERMISSION_ERROR : exception;
end OS.EXCEPTIONS;

with OS.EXCEPTIONS;
package OS.FILE_MANAGER is
    -- Define OS file operations
    type FILE_MODE is (READ_ONLY, WRITE_ONLY, READ_WRITE);

    function OPEN(
      NAME : STRING;
      MODE : FILE_MODE
    ) return OS.FILE_DESCRIPTOR;

    procedure CLOSE(FILE : in OS.FILE_DESCRIPTOR);

    -- etc...
end OS.FILE_MANAGER;
```

10.1.1. Context Clauses - With Clauses

A with clause may reference a child library unit. This necessitates an expanded name notation rather than a simple name notation in a context clause.

with_clause ::= **with** *library_unit*_name {, *library_unit*_name};

The expanded name used in a with clause must be a full expanded name of the library unit starting with a root library unit name. A with clause that includes an expanded name also includes an implicit ‘‘with’’ of the library unit named by each prefix of the expanded name. In other words, referencing a child library unit in a with clause by its expanded name is equivalent to referencing both the child library unit and its parent in the with clause.

Example:

```
with OS.FILE_MANAGER;   -- implies: with OS;
procedure MAIN is
    FD : OS.FILE_DESCRIPTOR;   -- from package OS
begin
    FD := OS.FILE_MANAGER.OPEN(
            "test_data", OS.FILE_MANAGER.READ_ONLY);
    . . .
end MAIN;
```

If a library unit named in a with clause is private, then its parent must be an ancestor of the compilation unit being defined. Furthermore, the declaration of a visible child library unit must not depend on a private child of the same parent package.

[*Note*: If a library unit is renamed at the library unit level, then the name introduced by the renaming may also be used in a with clause. If a child library unit is renamed as a root library unit, a ‘‘with’’ using this new simple name does not implicitly ‘‘with’’ its parent.]

10.1.2. Main Subprograms, Partitions, and Environment Tasks [*new*]

A program consists of one or more *partitions*. Each partition consists of one or more library units; one of the library unit subprograms of a partition may be designated as the *main subprogram* for that partition.

The library units that make up a partition are elaborated by the *environment task* of that partition, and then, if present, the body of the main subprogram is executed. Upon completion of the main subprogram (if any), the environment task waits until all tasks dependent on its library packages terminate. The environment task then leaves the scope of the library units of the partition, thereby terminating the partition.

An implementation must provide a mechanism for building a partition from a set of library units specified by the user. In the absence of a pragma specifying otherwise (see DS:ALL), a partition will also include a copy of all other library units on which the specified units depend.

A program terminates when all of its partitions have terminated, or in an implementation-defined manner.

[*Note*: A partition may be thought of as a separate address space, possibly running on a separate computer. Implementations are not required to use separate (hardware) address spaces for separate partitions.]

[*Note*: An implementation may provide inter-partition communication mechanism(s) via special packages and pragmas. Standard package categorization pragmas for distribution and methods for specifying inter-partition communication are defined in DS:ALL.]

10.1.3. Examples of Compilation Units [*originally 10.1.2*]

10.2. Subunits of Compilation Units

[*Note*: We have removed the restriction that all subunits of the same ancestor library unit must have distinct simple identifiers. Instead we require only that the full expanded names of all program units that are compilation units be unique. See 10.1.]

By analogy with task bodies, protected unit bodies may be compiled as subunits.

10.2.1. Examples of Subunits

10.3. Order of Compilation

10.4. The Program Library

The description of the program library will be less concrete, and will allow multiple program libraries, possibly taking advantage of the hierarchical library unit name space.

10.5. Elaboration of Library Units

As in Ada 83, library unit elaboration is performed in an order consistent with the partial ordering defined by with clauses (RM 10.5(2)). In addition, child library units must be elaborated after the specification of their parent package.

The syntax for pragma ELABORATE is revised to allow the specification of the full expanded name of a child library unit.

If the body of a library unit identified in a pragma ELABORATE depends on another library unit, then the pragma ELABORATE applies to this other unit as well (and so on, recursively). [*Note*: This ensures that a subprogram of a library unit identified in a pragma ELABORATE, may be called during the elaboration of the unit with the pragma, without raising PROGRAM_ERROR.]

[*Note*: An implementation may provide additional pragmas for controlling elaboration; Annex SP:ALL defines standard ones.]

[*Note*: We are studying a proposal to require that all access-before-elaboration (ABE) checking be performed statically (presumably at link-time in most cases). If the rules can be kept simple yet flexible, and the implementation is straightforward, this approach could be a big improvement over the complexity and inefficiency of run-time ABE checks.]

10.5.1. Library Unit Package Categorization [*new*]

[*Note*: Further categorizations of library units are discussed in the Annex to which they apply.]

Each library unit package may be *categorized* using a pragma. The categorization pragma must appear immediately following the **is** of the package specification. It applies to both the specification and body of the package, but not to any child library units. The following categorization pragmas are defined:

PURE A *pure* package must not contain library-level variable declarations, library-level access type declarations, or library-level declarations whose elaboration involves the execution of a program body. A pure package must depend only on other pure packages. It must not depend on any library unit subprogram. The body of a pure package must not have a sequence of statements. A library-level subpackage of a pure package must also be pure. All pure library unit packages are elaborated prior to other library units. [*Note*: These rules are intended to ensure that a library-level subprogram in a pure package need not perform an access-before-elaboration check.]

ELABORATE_BODY
The body of a package with the ELABORATE_BODY pragma is elaborated immediately after completing elaboration of the package specification. Any dependences of the package body are effectively dependences of the package specification, and must be elaborated prior to elaborating the package specification.

[*Note*: The above pragmas will allow compilers to eliminate certain access-before-elaboration checks. Pure packages are particularly important in environments lacking fully shared memory. See Annex DS:ALL for a further discussion of distributing programs.]

Package STANDARD and package SYSTEM are pure, as are certain packages defined within the Specialized Needs Annexes. All other language-defined packages are impure.

10.6. Program Optimization

11. Ada 9X Exceptions

11.1. Exception Declarations

11.2. Exception Handlers

The syntax for exception handlers is changed to allow a choice parameter. [*Note*: The definition of exception_handler_part is here so as to localize the presentation of the syntax. It does not represent a language change.]

```
exception_handler_part ::=
  exception exception_handler {exception_handler}

exception_handler ::=
  when [choice_parameter:] exception_choice {| exception_choice} =>
    sequence_of_statements

choice_parameter ::= identifier
```

The scope of a choice parameter is the exception handler in which it is declared. A choice parameter is a constant of type SYSTEM.EXCEPTION_OCCURRENCE; it contains information about the occurrence of the exception being handled.

The type EXCEPTION_OCCURRENCE is an implementation-defined non-limited type. [*Note*: Values of this type may be written to an error log for later analysis, or may be passed to subprograms for immediate error analysis.]

The following functions are defined in package SYSTEM:

```
function EXCEPTION_NAME(X: EXCEPTION_OCCURRENCE)
   return STRING;

function EXCEPTION_INFORMATION(X: EXCEPTION_OCCURRENCE)
   return STRING;
```

EXCEPTION_NAME returns a name of the exception. It is implementation-defined whether this returns the simple name of the exception, or the full, expanded name of the exception.

EXCEPTION_INFORMATION returns, as a string, any other information the implementation wishes to provide. This information should not include the name of the exception, but should include information identifying the location where the exception occurred, and, for predefined exceptions, the specific kind of run-time check that failed. The string returned should be in a form suitable for printing to an error log file.

[*Note*: The purpose of these functions is to allow programs to print out debugging/error-logging information in a portable way. A program that prints out the EXCEPTION_INFORMATION is portable in the sense that it will work in any implementation; it might print out different information, but the presumption is that the information printed out is appropriate for debugging/error analysis on that system.

It is preferred that EXCEPTION_NAME return the full expanded name, starting with the root library unit in which the exception is declared.

For a re-raise statement, it is preferred that the implementation not create a new EXCEPTION_OCCURRENCE, but instead propagate the same information. This allows the original cause of the exception to be determined.

Implementations are allowed to provide other operations.]

11.3. Raise Statements

11.4. Exception Handling

11.4.1. Exceptions Raised During the Execution of Statements

11.4.2. Exceptions Raised During the Elaboration of Declarations

11.5. Exceptions Raised During Task Communication

11.6. Exceptions and Optimization

[*Note*: The Safety/Security Annex will provide additional control over optimization and code generation. The ‘‘core’’ rules will be revised to ensure that vectorization and other kinds of pipelining and parallelism can be utilized to the maximum extent possible, unless a (very) local exception handler is present.]

11.7. Suppressing Checks

12. Ada 9X Generic Units

12.1. Generic Declarations

The kinds of generic formal parameters will be extended to include generic formal packages and generic formal derived types.

The specification of a formal parameter will distinguish between types that may be used to declare uninitialized objects, and those that must only be used to declare parameters or initialized objects.

```
generic_parameter_declaration ::=
  . . . (as before)
 |  type identifier[formal_discriminant_part]  is
     [tagged] [limited]  private;
 |  type identifier[formal_discriminant_part]  is
     new type_mark [with private];
 |  with package identifier  is new generic_package_name(<>);

formal_discriminant_part ::= discriminant_part | (<>)
```

As in Ada 83, within a generic unit, attributes and values of generic formal parameters are never static. However, within a generic declaration, other static semantic checks that depend on information not specified by the formal part are postponed until the generic is instantiated.

[*Note*: The checks on declarative items within a generic declaration that are postponed until instantiation include:

- the scope check associated with deriving from a tagged type (see 3.4), when the check depends on the nesting level of the instantiation;
- the scope check associated with use of the ACCESS attribute (see 3.10.2) or access type conversion (see 4.6), when the check depends on the nesting level of the instantiation;
- the check that a component name in a discriminant part or extension part of a derived type definition does not clash with the name of a component inherited on derivation (see 3.6.1 and 3.8.2), when the parent is a formal private or derived type, or derived from one;
- the check that a non-limited tagged type is not extended with a limited extension part (see 3.4.2);
- the check that a function with a controlling result is explicitly overridden when deriving from a tagged type (see 3.4.2).

These checks must be performed when the generic is instantiated.]

In a generic body, such checks are not postponed, to ensure that the legality of an instantiation will not depend on the generic body. Within the body, formal types are assumed to have the characteristics specified within the formal part:

- If a formal type is not tagged, then the type is treated as an untagged type within the generic body. Deriving from such a type in a generic body is permitted, and does not create a distinctly tagged type.
- If a formal type is private, then within the generic body, the type is considered private with the only named components being the discriminants specified in the formal discriminant part.

For checks that depend on other characteristics of the actual instantiation, the worst case is assumed, and the generic body is rejected if any legal instantiation might cause the check to fail.

[*Note*: Assuming the ‘‘worst’’ in a generic body implies the following:

- the scope checks are performed assuming the body will be instantiated at a level deeper than its declaration;
- the staticness checks are performed assuming the generic formal parameters are not static;
- it is not permitted to derive from a formal tagged type in a generic body.]

12.1.1. Generic Formal Objects

12.1.2. Generic Formal Types

Add the following paragraph:

(h) For a formal derived type declaration, the available operations are the primitive operations defined for the specified parent subtype.

For a formal derived type, these primitive operations are those of the parent type, even if the operation has been redefined or hidden for the actual type. [*Note*: This rule is necessary for untagged types because their primitive operations may be redefined or hidden by operations that are not subtype conformant with the parent’s operations (see RM 8.3(15)). For a tagged type, the primitive operations of the parent type are in any case defined as dispatching to the appropriate body based on the type of the actual parameters (see 3.4.3).]

For a formal private or derived type with (<>) as the formal discriminant part, the type must be assumed to have an unspecified set of discriminants without defaults, or be a tagged class-wide type, disallowing its use where a constrained type or a type with defaults for discriminants is required (see 3.6.1 and 3.7.2).

A generic formal private type is considered tagged within the generic only if the modifier **tagged** appears in the formal type declaration. In an instantiation, the actual type may be tagged even if the formal is not.

If the parent type of a formal derived type is tagged, then the reserved words **with private** must appear in the generic type definition, and the formal derived type is itself considered a tagged type. [*Note:* This preserves the property that a tagged type includes in its declaration either the reserved word **tagged** or the reserved word **with**.]

The actual type for a formal numeric type or a formal discrete type must not be.a non-standard numeric type (see 3.5.4 and 3.5.6). [*Note*: This restriction is necessary because non-standard numeric types have some number of restrictions on their use, which could cause ‘‘contract model’’ problems in a generic body. Note that such types can be used with formal derived and formal private types.]

12.1.3. Generic Formal Subprograms

12.1.4. Generic Formal Packages [*new*]

A generic parameter declaration that is in the form of a package instantiation declares a generic formal package.

A generic formal package denotes the package instantiation supplied as the corresponding generic actual parameter in a generic instantiation, as described in section 12.3. In addition, the actual parameters of the actual package instantiation are available using selected name notation.

Example of using a generic formal package parameter:

```
generic
    type FLOAT_TYPE is digits <>;
package GENERIC_COMPLEX_FUNCTIONS is
  -- This package defines a COMPLEX type, as a pair
  -- of FLOAT_TYPE values
    type COMPLEX is
      record
        REAL : FLOAT_TYPE;
        IMAG : FLOAT_TYPE;
      end record;

    -- Operations on the COMPLEX type
    function "-"(RIGHT : COMPLEX) return COMPLEX;
    function "+"(LEFT, RIGHT : COMPLEX) return COMPLEX;
    . . .
end GENERIC_COMPLEX_FUNCTIONS;

generic
    with package COMPLEX_FUNCTIONS is
      new GENERIC_COMPLEX_FUNCTIONS(<>);
package GENERIC_COMPLEX_MATRIX_OPERATIONS is
  -- This package defines a COMPLEX_MATRIX type, given an
  -- instantiation of the GENERIC_COMPLEX_FUNCTIONS package
    type COMPLEX_MATRIX is
      array(POSITIVE range <>, POSITIVE range <>)
        of COMPLEX_FUNCTIONS.COMPLEX;

    -- Operations on COMPLEX_MATRIX type and COMPLEX type.
    function "*"(LEFT : COMPLEX_FUNCTIONS.COMPLEX;
                 RIGHT : COMPLEX_MATRIX)
      return COMPLEX_MATRIX;
    . . .
end GENERIC_COMPLEX_MATRIX_OPERATIONS;

-- Here are a pair of instantiations:
package SHORT_COMPLEX_PKG is
  new GENERIC_COMPLEX_FUNCTIONS(SHORT_FLOAT);

. . .

package SHORT_COMPLEX_MATRIX_PKG is
  new GENERIC_COMPLEX_MATRIX_OPERATIONS(SHORT_COMPLEX_PKG);
```

Example of using generic formal package as a signature:

```
generic
   -- This generic defines a signature for a mathematical group
     type GROUP_ELEMENT is private;
     IDENTITY : in GROUP_ELEMENT;
     with function OP(LEFT, RIGHT : GROUP_ELEMENT)
       return GROUP_ELEMENT;
     with function INVERSE(RIGHT : GROUP_ELEMENT)
       return GROUP_ELEMENT;
package GROUP_SIGNATURE is end;

generic
     with package GROUP is new GROUP_SIGNATURE(<>);
function POWER(LEFT : GROUP.GROUP_ELEMENT; RIGHT : INTEGER)
  return GROUP.GROUP_ELEMENT;

function POWER(LEFT : GROUP.GROUP_ELEMENT; RIGHT : INTEGER)
  return GROUP.GROUP_ELEMENT is
    -- This generic function applies the group operation
    -- to the given group element the specified number of times.
    -- If the right operand is negative, the inverse of the result
    -- is returned; if it is zero, the identity is returned.
    RESULT : GROUP.GROUP_ELEMENT := IDENTITY;
begin
    for I in 1 .. abs RIGHT loop
        RESULT := GROUP.OP(RESULT, LEFT);
    end loop;
    if RIGHT < 0 then
        return GROUP.INVERSE(RESULT);
    else
        return RESULT;
    end if;
end POWER;

-- Here is an instantiation that ''claims'' that the
-- short complex numbers are a group over addition:
package SHORT_COMPLEX_ADDITION_GROUP is
  new GROUP_SIGNATURE(SHORT_COMPLEX_PKG.COMPLEX,
    IDENTITY => (0.0, 0.0),
    OP => SHORT_COMPLEX_PKG."+",
    INVERSE => SHORT_COMPLEX_PKG."-");

-- Here is an instantiation of POWER for the complex addition group:
function COMPLEX_MULTIPLICATION is
  new POWER(SHORT_COMPLEX_ADDITION_GROUP);
```

12.2. Generic Bodies

12.3. Generic Instantiation

Package instantiations are allowed as a new kind of generic parameter.

generic_actual_parameter ::=
  . . . *(as before)*
  | *package_instantiation*_name

12.3.1. Matching Rules for Formal Objects

12.3.2. Matching Rules for Formal Private Types

If the modifier **tagged** appears in a formal private type declaration, then the actual type must be a tagged type. The actual type must not be an abstract type (see 3.4.2).

Paragraph 3 of RM 12.3.2 is revised as follows:

- . . .[same first sentence]. If the formal discriminant part is (<>) then the actual type may be with or without discriminants, constrained or unconstrained. If the formal type has no formal discriminant part, then the actual subtype may not be an unconstrained array subtype, nor may it be an unconstrained subtype with discriminants unless all discriminants have defaults.

Paragraph 4 of RM 12.3.2 is deleted — it is unnecessary because we now require (<>) if the actual subtype is unconstrained without defaults.

[*Note*: This is not upward compatible. The work-around is to add the (<>) if unconstrained types are to be supported as actual parameters to the generic.]

12.3.3. Matching Rules for Formal Scalar Types

12.3.4. Matching Rules for Formal Array Types

12.3.5. Matching Rules for Formal Access Types

The access modifier **constant** must apply to the actual access type if and only if it appears in the formal access type definition. The access modifier **all** must apply to the actual access type if it appears in the formal access type definition. See 3.9. [*Note*: If no modifier appears in the formal access type definition, the actual type may be either a pool-specific or a general access-to-variable type.]

12.3.6. Matching Rules for Formal Derived Types [*new*]

A formal derived type is matched by an actual subtype only if the root of the actual base type is the same as the specified parent base type or is derived from it, directly or indirectly. The actual type must not be an abstract type (see 3.4.2).

If a formal type has no formal discriminant part, then the actual subtype must be constrained or have defaults for all discriminants. If the formal discriminant part is (<>), then the actual type may be with or without discriminants, constrained or unconstrained. If the formal discriminant part includes discriminant specifications, then the discriminants of the actual type must match the discriminant specifications of the formal type as defined for formal private types (see 12.3.2).

A tagged class-wide type is considered a type with an unknown number of discriminants, and hence only matches a formal type with a discriminant part that is (<>).

If a generic unit has a formal derived type without a formal discriminant part, the constraints on the actual subtype must statically match those of the formal parent subtype. [*Note*: We are still studying the general issue of static versus run-time subtype matching checks on generic instantiation and array conversion.]

Example of formal derived type:

```
    package RATIONAL_ARITHMETIC is
      -- This package defines a rational number type

        type RATIONAL is limited private;

        procedure ASSIGN(LEFT : out RATIONAL;
                         RIGHT : in RATIONAL);

        function "+"(LEFT, RIGHT : RATIONAL) return RATIONAL;

        . . .
    end RATIONAL_ARITHMETIC;

    with RATIONAL_ARITHMETIC; use RATIONAL_ARITHMETIC;
    with TEXT_IO;
    generic
      -- This package provides I/O for any derivative of the RATIONAL type
        type NUM is new RATIONAL;
    package RATIONAL_IO is
        procedure GET(FILE  : in TEXT_IO.FILE_TYPE;
                      ITEM  : out NUM;
                      WIDTH : in TEXT_IO.FIELD := 0);

        procedure PUT(FILE : in TEXT_IO.FILE_TYPE;
                      ITEM : in NUM;
                      FORE : in TEXT_IO.FIELD;
                      AFT  : in TEXT_IO.FIELD;
                      EXP  : in TEXT_IO.FIELD);

    end RATIONAL_IO;
```

12.3.7. Matching Rules for Formal Subprograms [*originally 12.3.6*]

As in Ada 83, mode conformance is required between the formal and actual subprogram (see MS-6.3.1).

12.3.8. Matching Rules for Formal Packages [*new*]

A formal package is matched by an actual package instantiation only if the actual package is an instantiation of the generic package specified in the formal package declaration.

12.4. Example of a Generic Package

13. Ada 9X Representation Clauses and Implementation-Dependent Features

13.1. Representation Clauses

13.2. Attribute Definition Clauses [*originally Length Clauses*]

We generalize the term *length clause* to *attribute definition clause* since not all of the attributes being specified are actually ‘‘lengths.’’ This also allows this syntax to be used within the Specialized Needs Annexes for other user-specifiable attributes.

```
attribute_definition_clause ::=
   for simple_name'identifier use simple_expression;
 | for simple_name'identifier use name;
```

The attribute identifier in an attribute definition clause must denote an attribute that is *user-specifiable*.

An attribute definition clause must either contain a simple expression of an appropriate type, or the name of an entity of an appropriate kind, corresponding to the kind of result the attribute yields.

The effect of an attribute definition clause is to define the attribute of the entity denoted by the simple name, to be the value of the simple expression, or the entity denoted by the name.

In the absence of an attribute definition clause, the attribute is determined by default, according to language-defined or implementation-defined rules. In general, the default for an attribute of an object, or of a subtype declared by a subtype declaration, is determined by the attribute of the subtype named in the subtype indication given in its declaration.

[*Note*: We are considering moving some of these to the Systems Programming Annex or eliminating them completely, so as to minimize the number of user-specifiable attributes described in the core.]

The following attributes are user-specifiable:

SIZE	The size in bits for a type, subtype, or object. The expression must be a static expression of some integer type. For a type or subtype, the size is a lower bound. For an object, the specified size must be obeyed exactly.
COMPONENT_SIZE	The size in bits for the component of an array type. The expression must be a static expression of some integer type.
ALIGNMENT	The alignment for a type, subtype, or object, in storage units. If the alignment is positive, then the address of an object must be a multiple of its alignment. If the alignment is zero, then the object is not necessarily aligned on a storage unit boundary. The expression must be a nonnegative static expression of some integer type.
STORAGE_SIZE	The number of storage units to be reserved for an access type's storage pool, or a task type, subtype, or object. The expression must be of some integer type. As in Ada 83, the access type must not be a derived type.
SMALL	The value to be used for *small* for a fixed point type that is not a derived type. The expression must be a static expression of some real type. [*Note*: The name of this attribute may change to MODEL_SMALL or MACHINE_SMALL.]

ADDRESS	The address of an object, interrupt entry, task body, subprogram body, or the body of a package that is a compilation unit. The expression must be of type SYSTEM.ADDRESS. [*Note*: As in Ada 83, the ADDRESS attribute may also be specified using an address clause (see 13.5).]

Implementations may identify further attributes that are user-specifiable. Additional user-specifiable attributes are identified in Chapter 14 and in the Specialized Needs Annexes (see SPECIALIZEDNEEDS).

13.3. Enumeration Representation Clauses

13.4. Record Representation Clauses

The order, position, and size of the components of the parent part of a derived type may not be altered if the parent type is tagged or limited. Only the components of the extension part may have component clauses. No component of the extension part may be placed prior to a component of the parent part. [*Note*: For tagged, this allows assignment and equality on (view) conversions (see 4.6) of a tagged object to be performed using block operations. For limited, this ensures that conversion can be performed without copying; copying is inconsistent with having parts that might be controlled, protected, or inherently aliased.]

13.5. Address Clauses

If an address is specified for a subprogram or object that has a pragma IMPORT without a specified link name (see 13.9) the address expression need not be a link-time constant. [*Note*: Such an address clause is a portable alternative to an unchecked conversion from the address expression to an access-to-subprogram or object.]

An object with a user-specified address is considered an aliased object.

Note: The specification of a link name in a pragma EXPORT (see 13.9) for a subprogram or object is an alternative to explicit specification of its link-time address, allowing a link-time directive to place the subprogram or object within memory.

[*Note*: The attribute definition clause (see 13.2) may be used to specify the ADDRESS attribute, making this special syntax for address clauses possibly obsolescent.]

13.5.1. Interrupts

The support for interrupt handling is described in SP:ALL.

13.6. Change of Representation

13.7. The Package SYSTEM

The pragmas SYSTEM_NAME, STORAGE_UNIT, and MEMORY_SIZE are no longer required. However, the corresponding declarations in package SYSTEM are still required.

The following new declarations will be added to package SYSTEM:

```
package SYSTEM is

   -- . . . As in Ada 83

   -- Define a type and subtypes for offsets in storage units.
   type STORAGE_OFFSET is range implementation-defined;
   subtype STORAGE_COUNT is STORAGE_OFFSET
     range 0..STORAGE_OFFSET'LAST;
   subtype STORAGE_INDEX is STORAGE_OFFSET
     range 1..STORAGE_OFFSET'LAST;

   -- Define an integer type and an array type
   -- for representing storage elements
   type STORAGE_ELEMENT is range 0..2**STORAGE_UNIT-1;
   type STORAGE_ARRAY is array
     (STORAGE_INDEX range <>) of STORAGE_ELEMENT;
   pragma PACK(STORAGE_ARRAY);

   -- Define operators for doing address arithmetic and comparison
   function "+"(LEFT : ADDRESS; RIGHT : STORAGE_OFFSET)
     return ADDRESS;
   function "+"(LEFT : STORAGE_OFFSET; RIGHT : ADDRESS)
     return ADDRESS;
   function "-"(LEFT : ADDRESS; RIGHT : STORAGE_OFFSET)
     return ADDRESS;
   function "-"(LEFT, RIGHT : ADDRESS)
     return STORAGE_OFFSET;

   function "<" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function "<="(LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function ">" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function ">="(LEFT, RIGHT : ADDRESS) return BOOLEAN;
end SYSTEM;
```

The semantics of the arithmetic and comparison operators on ADDRESS are implementation-defined. In particular, an implementation on a segmented machine may raise CONSTRAINT_ERROR on certain inter-segment operations.

13.7.1. System-Dependent Named Numbers

13.7.2. Representation Attributes

[*Note*: There will be more specification for the meaning of the ADDRESS and SIZE attributes, including in the presence of unconstrained or dynamically-constrained objects.]

13.7.3. Representation Attributes of Real Types

[*Note*: See the Numerics Annex, NM:ALL.]

13.8. Machine Code Insertions

13.9. Interface to Other Languages

The following pragmas are available for interfacing to other languages and controlling the overall representation of program entities, instead of the INTERFACE pragma:

IMPORT(*entity*_simple_name, *language*_simple_name [, string])

Specify that the entity denoted by the name is defined externally. The language simple name determines what calling convention (see 6.3.1) or layout convention is to be assumed for the entity. The elaboration of the declaration for such an entity has no effect other than to allow the identifier to denote the external entity. If such an entity is an object, no initial expression may be provided in its declaration. The optional third parameter specifies the *link name*, the name used to link the entity with the external environment. In the absence of such a link name, either an address clause must be specified for the entity, or a link name derived from its Ada name (in an implementation-defined manner) will be used. The pragma IMPORT must appear immediately within the same declarative region as the entity's declaration, prior to a forcing occurrence for the entity. It is implementation-defined what kinds of entities may be imported, and from what languages. In the presence of multiple implementations of the same language, the language simple name identifies both the language and the particular implementation.

EXPORT(*entity*_simple_name, *language*_simple_name [, string])

Specify that the (library-level) entity denoted by the name is to be accessible externally. The language simple name determines what calling convention (see 6.3.1) or layout convention is to be used for the entity. This pragma allows a non-Ada program to call an Ada subprogram, or reference an Ada object, for languages supported by an implementation. The optional third parameter specifies the link name for the entity. In the absence of such a link name, a link name derived from its Ada name (in an implementation-defined manner) will be used. The pragma EXPORT must appear immediately within the same declarative region as the entity's declaration, prior to a forcing occurrence for the entity. It is implementation-defined what kinds of entities may be exported, and what language conventions are supported. In the presence of multiple implementations of the same language, the language simple name identifies both the language and the particular implementation. [*Note*: Additional pragmas for further specifying the calling or layout conventions, perhaps on a parameter-by-parameter or component-by-component basis, are implementation-defined.]

LANGUAGE(*entity*_simple_name, *language*_simple_name)

Specify that the given entity should follow the calling convention (see 6.3.1) or layout conventions associated with the specified language. This pragma allows access-to-subprogram types to designate subprograms with non-Ada calling conventions. It also allows types to be layed out according to the conventions of some other language (e.g. COBOL or C) without requiring a detailed representation clause. The pragma LANGUAGE must appear immediately within the same declarative region as the entity's declaration, prior to a forcing occurrence for the entity. It is implementation-defined what language conventions are supported for which kinds of entities. In the presence of multiple implementations of the same language, the language simple name identifies both the language and the particular implementation.

13.10. Unchecked Programming

13.10.1. Unchecked Storage Deallocation

13.10.2. Unchecked Type Conversions

13.10.3. Unchecked Access Value Creation

The attribute P’UNCHECKED_ACCESS, when applied to a prefix P that denotes an aliased object, creates an access value for that object. The type of P’UNCHECKED_ACCESS must be determined from context, and must be an access type that could designate the object denoted by P if it had been declared at the library-level. If the type is an access parameter type (see 3.9), then the the scope level associated with the object designated by the access parameter will be library-level, for the purposes of subsequent type conversion scope checks (see 4.6).

All rules that apply to the ACCESS attribute (see 3.9.2), other than scope-checking rules, also apply to UNCHECKED_ACCESS. In other words, UNCHECKED_ACCESS bypasses scope checking, but still enforces other type checking rules in producing an access value.

[*Note*: This attribute is provided to support the situation where a local aliased object is to be inserted into a global linked data structure, when the programmer knows that it will always be removed from the data structure prior to exiting the object’s scope. ’ACCESS would be illegal in this case (see 3.10.2)]

14. Ada 9X Input-Output

14.1. External Files and File Objects

14.2. Sequential and Direct Files

14.2.1. File Management

14.2.2. Sequential Input-Output

The type SEQUENTIAL_IO.FILE_MODE will have a new enumeration literal APPEND_FILE, which indicates that the file should be opened for output and positioned at the end.

[*Note*: This is not upward compatible. For example, case statements on the mode will have to change. However, the incompatible cases should be rare, and the workaround is straightforward.]

14.2.3. Specification of the Package Sequential_IO

14.2.4. Direct Input-Output

[*Note*: APPEND_FILE mode is *not* supported for package DIRECT_IO.]

14.2.5. Specification of the Package Direct_IO

14.3. Text Input-Output

14.3.1. File Management

APPEND_FILE mode will be added as for SEQUENTIAL_IO.

14.3.2. Default Input and Output Files

14.3.3. Specification of Line and Page Lengths

14.3.4. Operations on Columns, Lines, and Pages

14.3.5. Get and Put Procedures

14.3.6. Input-Output of Characters and Strings

[*Note*: In accordance with the April ’92 ISO WG9 meeting, we have dropped proposals that depend on untagged class-wide types.]

14.3.7. Input-Output for Integer Types

[*Note*: In accordance with the April ’92 ISO WG9 meeting, we have dropped proposals that depend on untagged class-wide types.]

14.3.8. Input-Output for Real Types

[*Note*: In accordance with the April ’92 ISO WG9 meeting, we have dropped proposals that depend on untagged class-wide types.]

14.3.9. Input-Output for Enumeration Types

14.3.10. Specification of the Package Text_IO

14.4. Exceptions in Input-Output

14.5. Specification of the Package IO_Exceptions

14.6. Low Level Input-Output

This section and the package it describes is removed from the standard.

14.7. Stream Input-Output [*new*]

A *storage element* is a sequence of bits of length SYSTEM.STORAGE_UNIT. A *stream* is logically a sequence of storage elements. A stream type is a type in the class rooted at STREAM_SUPPORT.ROOT_STREAM_TYPE. A stream type may be implemented in various ways. Examples include an external file, an internal buffer, and a network channel.

Ada types for storage elements and streams are declared in the predefined STREAM_SUPPORT package. STREAM_SUPPORT.STREAM_ACCESS is an access type designating the class-wide type ROOT_STREAM_TYPE’CLASS.

14.7.1. T’READ and T’WRITE [*new*]

This section specifies operations for converting a value of a type into a stream of storage elements and reconstructing a value of the type from such a stream.

The T’READ and T’WRITE operations are user-specifiable attributes. All non-limited types have default implementations for these operations. The default implementations may be overridden, as follows:

```
procedure WRITE(
  STREAM : STREAM_SUPPORT.STREAM_ACCESS; ITEM: T);
for T'WRITE use WRITE;

function READ(STREAM : STREAM_SUPPORT.STREAM_ACCESS)
  return  T;
for T'READ use READ;
```

The T'WRITE operation converts a value of the type T to a sequence of storage elements, and appends this sequence to the specified stream.

The T'READ operation removes a sequence of storage elements from the specified stream, converts it to a value of type T, and returns the value. It expects the next sequence of storage elements for that stream to have the form written by T'WRITE; if the elements do not have that form, CONSTRAINT_ERROR is raised.

14.7.2. Predefined T'WRITE and T'READ [*new*]

The predefined T'READ and T'WRITE operations for an elementary type T transfer the number of storage elements that would be used for a component of an unpacked array of T.

The predefined T'READ and T'WRITE for a composite type execute as follows:

- For an array type T defined by an unconstrained array definition, the length of each dimension appears first in the stream, followed by the components.

  The representation of the length of a dimension is the same as the representation that would be used by the implementation for a value of an integer type whose range is 0 up to the maximum length for the dimension.

- For a type T that is packed or has a representation clause, the components of the type are transferred in a block using the in-memory representation of the composite type.

- For a type T that is not packed and has no representation clause, the components of the type are transferred by sequentially executing the READ or WRITE operation for each component type, in the canonical component order. This order is last dimension varying fastest for an array, and positional-aggregate order for a record.

  If two consecutive components of such a type do not have compatible alignments, then a call on the stream's ALIGN operation is performed between the READ or WRITE operations of the two components. The ALIGN operation is passed the alignment requirements of the second of the two components. A given stream type may choose to ignore the call on ALIGN, or it may use it to keep the stream aligned in a way that is compatible with the alignment requirements of the objects in the stream.

14.7.3. Validation on T'READ [*new*]

The predefined T'READ operation for a scalar type checks that the value is within the range of the base type. CONSTRAINT_ERROR is raised if this check fails.

The predefined T'READ operation for a composite type must omit range checks on scalar subcomponents (other than discriminants) that do not have a default initial expression in their component declaration, as such components need not be initialized prior to being written with T'WRITE.

14.7.4. T'WRITE and T'READ for Tagged Class-Wide Types [*new*]

The predefined READ and WRITE operations for a tagged class-wide type T'CLASS use a unique value to represent the type tag. The value used to represent each tag is determined at the time the corresponding type declaration is compiled. For each tagged type, the value used to represent the tag is distinct from the value used to represent the tags of other previously compiled types in that class. If T'CLASS'READ does not recognize the value as a valid tag identifier for a type included in the program, it raises CONSTRAINT_ERROR.

Notes:

T'READ and T'WRITE will generally be implemented in terms of READ and WRITE operations of simpler component types, or in terms of the primitive stream operations defined in package STREAM_SUPPORT.

14.7.5. The Package STREAM_SUPPORT [*new*]

ROOT_STREAM_TYPE is the root type of the class of stream types. The types in this class represent different kinds of streams. A new type is derived to represent the structure and requirements of each kind of stream. READ, WRITE, and ALIGN operations must be provided on each new stream type. These operations are used by the predefined T'READ and T'WRITE operations. They may also be used by user-defined T'READ and T'WRITE operations.

```
with SYSTEM;
package STREAM_SUPPORT is

    type ROOT_STREAM_TYPE is tagged limited private;
    type STREAM_ACCESS is access ROOT_STREAM_TYPE'CLASS;

    procedure READ(
      STREAM : in out ROOT_STREAM_TYPE;
      ITEM : out SYSTEM.STORAGE_ARRAY;
      LAST : out SYSTEM.STORAGE_COUNT) is <>;
    procedure WRITE(
      STREAM : in out ROOT_STREAM_TYPE;
      ITEM : in SYSTEM.STORAGE_ARRAY) is <>;
    procedure ALIGN(
      STREAM : in out ROOT_STREAM_TYPE;
      ALIGNMENT : in SYSTEM.STORAGE_COUNT) is <>;

private
    . . . -- implementation defined
end STREAM_SUPPORT;
```

The READ operation transfers ITEM'LENGTH storage elements from the specified stream to fill the array ITEM. The index of the last storage element transferred is returned in LAST. LAST is only less than ITEM'LAST if the end of the stream is reached.

The WRITE operation appends ITEM to the specified stream.

If the stream maintains alignment, then the ALIGN operation appends the minimum number of storage items to the specified stream needed to cause the position within the stream to be a multiple of ALIGNMENT.

14.7.6. Heterogeneous Input-Output [*new*]

A general stream-oriented I/O package is provided to support heterogeneous I/O. All of the operations declared here that correspond to SEQUENTIAL_IO operations have the same semantics as those SEQUENTIAL_IO operations.

```
with IO_EXCEPTIONS;
package STREAM_SUPPORT.STREAM_IO is

  -- This package is similar to Sequential_IO,
  -- Except that it reads and writes Storage_Elements
  -- directly or by using T'READ and T'WRITE

    type FILE_TYPE is limited private;
    type FILE_MODE is (IN_FILE, OUT_FILE, APPEND_FILE);

    procedure CREATE(FILE : in out FILE_TYPE;
                     MODE : in FILE_MODE := OUT_FILE;
                     NAME : in STRING := "";
                     FORM : in STRING := "");

    procedure OPEN(FILE : in out FILE_TYPE;
                   MODE : in FILE_MODE;
                   NAME : in STRING;
                   FORM : in STRING := "");
```

```
procedure CLOSE(FILE : in out FILE_TYPE);
procedure DELETE(FILE : in out FILE_TYPE);
procedure RESET(FILE : in out FILE_TYPE; MODE : in FILE_MODE);
procedure RESET(FILE : in out FILE_TYPE);

function MODE(FILE : in FILE_TYPE) return FILE_MODE;
function NAME(FILE : in FILE_TYPE) return STRING;
function FORM(FILE : in FILE_TYPE) return STRING;

function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

function STREAM(FILE : in FILE_TYPE)
  return STREAM_SUPPORT.STREAM_ACCESS;
    -- Return stream access for use with T'WRITE and T'READ.

-- The following operations are equivalent
-- to the analogous operations on Streams:

-- Read array of storage elements from file
procedure READ(
  FILE : in FILE_TYPE;
  ITEM : out SYSTEM.STORAGE_ARRAY;
  LAST : out SYSTEM.STORAGE_COUNT
  -- Last < ITEM'LAST only if end-of-stream reached
);   -- Raises USE_ERROR if mode is not In_File

-- Write array of storage elements into file
procedure WRITE(
  FILE : in FILE_TYPE;
  ITEM : in SYSTEM.STORAGE_ARRAY
);   -- Raises USE_ERROR if mode is not Out_File or Append_File

-- Align file offset to be multiple of alignment
procedure ALIGN(
  FILE : in FILE_TYPE;
  ALIGNMENT : in SYSTEM.STORAGE_COUNT
);   -- Adjust offset in File to multiple of alignment
```

```
    -- Operations on position within file
    -- These operations raise USE_ERROR if the file
    -- does not support repositioning.

    type FILE_OFFSET is range 0..<implementation_DEFINED>;
      -- Offset in file, in storage units.

    type BASE_POSITION is
      (BEGINNING_OF_FILE, CURRENT_POSITION, END_OF_FILE);
      -- Base for repositioning

    -- Get current file position,
    -- as an offset relative to beginning of file
    function POSITION(FILE : in FILE_TYPE) return FILE_OFFSET;

    -- Set file position
    procedure SET_POSITION(
      FILE : in FILE_TYPE;
      OFFSET : in FILE_OFFSET;
      RELATIVE_TO : BASE_POSITION := BEGINNING_OF_FILE
    );

    -- Get current size, in storage units
    function SIZE(FILE : in FILE_TYPE) return FILE_OFFSET;

    -- Change mode without repositioning
    procedure SET_MODE(FILE : in out FILE_TYPE; MODE : in FILE_MODE);

    -- Flush any buffers associated with file,
    -- without closing file or repositioning
    procedure FLUSH(FILE : in out FILE_TYPE);

    -- exceptions
    STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR : exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR : exception renames IO_EXCEPTIONS.USE_ERROR;
    DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
    END_ERROR : exception renames IO_EXCEPTIONS.END_ERROR;
    DATA_ERROR : exception renames IO_EXCEPTIONS.DATA_ERROR;

end STREAM_SUPPORT.STREAM_IO;
```

14.8. Example of Input-Output

Index

Table of Contents