INTERNATIONAL ORGANIZATION FOR STANDARDIZATION

INTERNATIONAL ELECTROTECHNICAL COMMISSION

Changes to Ada — 1987 to 1995

Language and Standard Libraries

Version 5.95
25 November 1994

Contents

Foreword

This document lists in detail the changes introduced in the second (Ada 9X) edition of the Ada standard (ISO/IEC 8652:1995) with respect to the first (Ada 83) edition (ISO 8652:1987).

Information technology — Programming Languages — Ada —

Section 1: General

1.1 Scope

1.1.1 Extent

1.1.2 Structure

Inconsistencies With Ada 83

This heading lists all of the upward inconsistencies between Ada 83 and Ada 9X. Upward inconsistencies are situations in which a legal Ada 83 program is a legal Ada 9X program with different semantics. This type of upward incompatibility is the worst type for users, so we only tolerate it in rare situations. 0.a

(Note that the semantics of a program is not the same thing as the behavior of the program. Because of Ada's indeterminacy, the ''semantics'' of a given feature describes a *set* of behaviors that can be exhibited by that feature. The set can contain more than one allowed behavior. Thus, when we ask whether the semantics changes, we are asking whether the set of behaviors changes.) 0.b

This is not part of the definition of the language, and does not appear in the RM9X. 0.c

Incompatibilities With Ada 83

This heading lists all of the upward incompatibilities between Ada 83 and Ada 9X, except for the ones listed under ''Inconsistencies With Ada 83'' above. These are the situations in which a legal Ada 83 program is illegal in Ada 9X. We do not generally consider a change that turns erroneous execution into an exception, or into an illegality, to be upwardly incompatible. 0.d

This is not part of the definition of the language, and does not appear in the RM9X. 0.e

Extensions to Ada 83

This heading is used to list all upward compatible language changes; that is, language extensions. These are the situations in which a legal Ada 9X program is not a legal Ada 83 program. The vast majority of language changes fall into this category. 0.f

0.g This is not part of the definition of the language, and does not appear in the RM9X.

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

Wording Changes From Ada 83

0.i This heading lists some of the non-semantic changes between RM83 and the RM9X. It is incomplete; we have not attempted to list all wording changes, but only the ‘‘interesting’’ ones.

0.j This is not part of the definition of the language, and does not appear in the RM9X.

1.1.3 Conformity of an Implementation with the Standard

1.1.4 Method of Description and Syntax Notation

1.1.5 Classification of Errors

Wording Changes From Ada 83

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

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

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

1.2 Normative References

1.3 Definitions

Section 2: Lexical Elements

2.1 Character Set

Extensions to Ada 83

Ada 9X allows 8-bit and 16-bit characters, as well as implementation-specified character sets. 0.a

Wording Changes From Ada 83

The syntax rules in this clause are modified to remove the emphasis on basic characters vs. others. (In this day and age, there is no need to point out that you can write programs without using (for example) lower case letters.) In particular, character (representing all characters usable outside comments) is added, and basic_graphic_character, other_special_character, and basic_character are removed. Special_character is expanded to include Ada 83's other_special_character, as well as new 8-bit characters not present in Ada 83. Note that the term ‘‘basic letter’’ is used in A.3, ‘‘Character Handling’’ to refer to letters without diacritical marks. 0.b

Character names now come from ISO 10646. 0.c

We use identifier_letter rather than letter since ISO 10646 BMP includes many "letters’ that are not permitted in identifiers (in the standard mode). 0.d

2.2 Lexical Elements, Separators, and Delimiters

2.3 Identifiers

Wording Changes From Ada 83

We no longer include reserved words as identifiers. This is not a language change. In Ada 83, identifier included reserved words. However, this complicated several other rules (for example, regarding implementation-defined attributes and pragmas, etc.). We now explicitly allow certain reserved words for attribute designators, to make up for the loss. 0.a

2.4 Numeric Literals

2.4.1 Decimal Literals

Wording Changes From Ada 83

We have changed the syntactic category name integer to be numeral. We got this idea from ACID. It avoids the confusion between this and integers. (Other places don't offer similar confusions. For example, a string_literal is different from a string.) 0.a

2.4.2 Based Literals

Wording Changes From Ada 83

The rule about which letters are allowed is now encoded in BNF, as suggested by Mike Woodger. This is clearly more readable. 0.a

2.5 Character Literals

Wording Changes From Ada 83

The definitions of the values of literals are in Sections 3 and 4, rather than here, since it requires knowledge of types. 0.a

2.6 String Literals

Wording Changes From Ada 83

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

0.b The syntax is described differently.

2.7 Comments

2.8 Pragmas

Incompatibilities With Ada 83

0.a In Ada 83, ''bad'' pragmas are ignored. In Ada 9X, they are illegal, except in the case where the name of the pragma itself is not recognized by the implementation.

Extensions to Ada 83

0.b Implementation-defined pragmas may affect the legality of a program.

Wording Changes From Ada 83

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

Extensions to Ada 83

0.d The Optimize pragma now allows the identifier Off to request that normal optimization be turned off.

0.e An Optimize pragma may appear anywhere pragmas are allowed.

Wording Changes From Ada 83

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

2.9 Reserved Words

Incompatibilities With Ada 83

The following words are not reserved in Ada 83, but are reserved in Ada 9X: **abstract**, **aliased**, **protected**, **requeue**, **tagged**, **until**. 0.a

Wording Changes From Ada 83

The clause entitled ‘‘Allowed Replacements of Characters’’ has been moved to Annex J, ‘‘Obsolescent Features’’. 0.b

Chg839X;5.95

Section 3: Declarations and Types

3.1 Declarations

Wording Changes From Ada 83

The syntax rule for defining_identifier is new. It is used for the defining occurrence of an identifier. Usage occurrences use the direct_name or selector_name syntactic categories. Each occurrence of an identifier (or simple_name), character_literal, or operator_symbol in the Ada 83 syntax rules is handled as follows in Ada 9X: 0.a

- It becomes a defining_identifier, defining_character_literal, or defining_operator_symbol (or some syntactic category composed of these), to indicate a defining occurrence; 0.b
- It becomes a direct_name, in usage occurrences where the usage is required (in Section 8) to be directly visible; 0.c
- It becomes a selector_name, in usage occurrences where the usage is required (in Section 8) to be visible but not necessarily directly visible; 0.d
- It remains an identifier, character_literal, or operator_symbol, in cases where the visibility rules do not apply (such as the designator that appears after the **end** of a subprogram_body). 0.e

For declarations that come in ‘‘two parts’’ (program unit declaration plus body, private or incomplete type plus full type, deferred constant plus full constant), we consider both to be defining occurrences. Thus, for example, the syntax for package_body uses defining_identifier after the reserved word **body**, as opposed to direct_name. 0.f

The defining occurrence of a statement name is in its implicit declaration, not where it appears in the program text. Considering the statement name itself to be the defining occurrence would complicate the visibility rules. 0.g

The phrase ‘‘visible by selection’’ is not used in Ada 9X. It is subsumed by simply ‘‘visible’’ and the Name Resolution Rules for selector_names. 0.h

(Note that in Ada 9X, a declaration is visible at all places where one could have used a selector_name, not just at places where a selector_name was actually used. Thus, the places where a declaration is directly visible are a subset of the places where it is visible. See Section 8 for details.) 0.i

We use the term ‘‘declaration’’ to cover _specifications that declare (views of) objects, such as parameter_specifications. In Ada 83, these are referred to as a ‘‘form of declaration,’’ but it is not entirely clear that they are considered simply ‘‘declarations.’’ 0.j

RM83 contains an incomplete definition of "elaborated" in this clause: it defines "elaborated" for declarations, declarative_parts, declarative_items and compilation_units, but "elaboration" is defined elsewhere for various other constructs. To make matters worse, Ada 9X has a different set of elaborable constructs. Instead of correcting the list, it is more maintainable to refer to the term "elaborable," which is defined in a distributed manner. 0.k

RM83 uses the term ‘‘has no other effect’’ to describe an elaboration that doesn’t do anything except change the state from not-yet-elaborated to elaborated. This was a confusing wording, because the answer to ‘‘other than what?’’ was to be found many pages away. In Ada 9X, we change this wording to ‘‘has no effect’’ (for things that truly do nothing at run time), and ‘‘has no effect other than to establish that so-and-so can happen without failing the Elaboration_Check’’ (for things where it matters). 0.l

We make it clearer that the term "execution" covers elaboration and evaluation as special cases. This was implied in RM83. For example, "erroneous execution" can include any execution, and RM83-9.4(3) has, "The task designated by any other task object depends on the master whose execution creates the task object;" the elaboration of the master’s declarative_part is doing the task creation. 0.m

3.2 Types and Subtypes

Wording Changes From Ada 83

This clause and its subclauses now precede the clause and subclauses on objects and named numbers, to cut down on the number of forward references. 0.a

We have dropped the term "base type" in favor of simply "type" (all types in Ada 83 were "base types" so it wasn’t clear when it was appropriate/necessary to say "base type"). Given a subtype S of a type T, we call T the "type of the subtype S." 0.b

3.2.1 Type Declarations

Wording Changes From Ada 83

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

0.b We have generalized the concept of first-named subtype (now called simply ‘‘first subtype’’) to cover all kinds of types, for uniformity of description elsewhere. RM83 defined first-named subtype in Section 13. We define first subtype here, because it is now a more fundamental concept. We renamed the term, because in Ada 9X some first subtypes have no name.

0.c We no longer elaborate discriminant_parts, because there is nothing to do, and it was complex to say that you only wanted to elaborate it once for a private or incomplete type. This is also consistent with the fact that subprogram specifications are not elaborated (neither in Ada 83 nor in Ada 9X). Note, however, that an access_definition appearing in a discriminant_part is elaborated when an object with such a discriminant is created.

3.2.2 Subtype Declarations

Incompatibilities With Ada 83

0.a In Ada 9X, all range_constraints cause freezing of their type. Hence, a type-related representation item for a scalar type has to precede any range_constraints whose type is the scalar type.

Wording Changes From Ada 83

0.b Subtype_marks allow only subtype names now, since types are never directly named. There is no need for RM83-3.3.2(3), which says a subtype_mark can denote both the type and the subtype; in Ada 9X, you denote an unconstrained (base) subtype if you want, but never the type.

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

3.2.3 Classification of Operations

Incompatibilities With Ada 83

0.a The attribute S’Base is no longer defined for non-scalar subtypes. Since this was only permitted as the prefix of another attribute, and there are no interesting non-scalar attributes defined for an unconstrained composite or access subtype, this should not affect any existing programs.

Extensions to Ada 83

0.b The primitive subprograms (derivable subprograms) include subprograms declared in the private part of a package specification as well, and those that override implicitly declared subprograms, even if declared in a body.

Wording Changes From Ada 83

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

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

3.3 Objects and Named Numbers

Extensions to Ada 83

0.a There are additional kinds of objects (choice parameters and entry indices of entry bodies).

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

Wording Changes From Ada 83

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

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

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

We have moved the syntax for object_declaration and number_declaration down into their respective subclauses, to keep the syntax close to the description of the associated semantics. 0.f

We talk about variables and constants here, since the discussion is not specific to object_declarations, and it seems better to have the list of the kinds of constants juxtaposed with the kinds of objects. 0.g

We no longer talk about indirect updating due to parameter passing. Parameter passing is handled in 6.2 and 6.4.1 in a way that there is no need to mention it here in the definition of read and update. Reading and updating now includes the case of evaluating or assigning to an enclosing object. 0.h

3.3.1 Object Declarations

Extensions to Ada 83

The syntax rule for object_declaration is modified to allow the **aliased** reserved word. 0.a

A variable declared by an object_declaration can be constrained by its initial value; that is, a variable of a nominally unconstrained array subtype, or discriminated type without defaults, can be declared so long as it has an explicit initial value. In Ada 83, this was permitted for constants, and for variables created by allocators, but not for variables declared by object_declarations. This is particularly important for tagged class-wide types, since there is no way to constrain them explicitly, and so an initial value is the only way to provide a constraint. It is also important for generic formal private types with unknown discriminants. 0.b

We now allow an unconstrained_array_definition in an object_declaration. This allows an object of an anonymous array type to have its bounds determined by its initial value. This is for uniformity: If one can write ‘‘X: **constant array**(Integer **range** 1..10) **of** Integer := ...;’’ then it makes sense to also allow ‘‘X: **constant array**(Integer **range** <>) **of** Integer := ...;’’. (Note that if anonymous array types are ever sensible, a common situation is for a table implemented as an array. Tables are often constant, and for constants, there’s usually no point in forcing the user to count the number of elements in the value.) 0.c

Wording Changes From Ada 83

We have moved the syntax for object_declarations into this subclause. 0.d

Deferred constants no longer have a separate syntax rule, but rather are incorporated in object_declaration as constants declared without an initialization expression. 0.e

3.3.2 Number Declarations

Extensions to Ada 83

We now allow a static expression of any numeric type to initialize a named number. For integer types, it was possible in Ada 83 to use ’Pos to define a named number, but there was no way to use a static expression of some non-universal real type to define a named number. This change is upward compatible because of the preference rule for the operators of the root numeric types. 0.a

Wording Changes From Ada 83

We have moved the syntax rule into this subclause. 0.b

AI-00263 describes the elaboration of a number declaration in words similar to that of an object_declaration. However, since there is no expression to be evaluated and no object to be created, it seems simpler to say that the elaboration has no effect. 0.c

3.4 Derived Types and Classes

Inconsistencies With Ada 83

When deriving from a (nonprivate, nonderived) type in the same visible part in which it is defined, if a predefined operator had been overridden prior to the derivation, the derived type will inherit the user-defined operator rather than the predefined operator. The work-around (if the new behavior is not the desired behavior) is to move the definition of the derived type prior to the overriding of any predefined operators. 0.a

Incompatibilities With Ada 83

When deriving from a (nonprivate, nonderived) type in the same visible part in which it is defined, a primitive subprogram of the parent type declared before the derived type will be inherited by the derived type. This can cause upward incompatibilities in cases like this: 0.b

0.c
```
            package P is
               type T is (A, B, C, D);
               function F( X : T := A ) return Integer;
               type NT is new T;
               -- inherits F as
               -- function F( X : NT := A ) return Integer;
               -- in Ada 9X only
               ...
            end P;
            ...
            use P;   -- Only one declaration of F from P is use-visible in
                     -- Ada 83;  two declarations of F are use-visible in
                     -- Ada 9X.
         begin
            ...
            if F > 1 then ... -- legal in Ada 83, ambiguous in Ada 9X
```

Extensions to Ada 83

0.d The syntax for a derived_type_definition is amended to include an optional record_extension_part (see 3.9.1).

0.e A derived type may override the discriminants of the parent by giving a new discriminant_part.

0.f The parent type in a derived_type_definition may be a derived type defined in the same visible part.

0.g When deriving from a type in the same visible part in which it is defined, the primitive subprograms declared prior to the derivation are inherited as primitive subprograms of the derived type. See 3.2.3.

Wording Changes From Ada 83

0.h We now talk about the classes to which a type belongs, rather than a single class.

0.i As explained in Section 13, the concept of "storage pool" replaces the Ada 83 concept of "collection." These concepts are similar, but not the same.

3.4.1 Derivation Classes

3.5 Scalar Types

Incompatibilities With Ada 83

0.a S'Base is no longer defined for nonscalar types. One conceivable existing use of S'Base for nonscalar types is S'Base'Size where S is a generic formal private type. However, that is not generally useful because the actual subtype corresponding to S might be a constrained array or discriminated type, which would mean that S'Base'Size might very well overflow (for example, S'Base'Size where S is a constrained subtype of String will generally be 8 * (Integer'Last + 1)). For derived discriminated types that are packed, S'Base'Size might not even be well defined if the first subtype is constrained, thereby allowing some amount of normally required ‘‘dope’’ to have been squeezed out in the packing. Hence our conclusion is that S'Base'Size is not generally useful in a generic, and does not justify keeping the attribute Base for nonscalar types just so it can be used as a prefix.

Extensions to Ada 83

0.b The attribute S'Base for a scalar subtype is now permitted anywhere a subtype_mark is permitted. S'Base'First .. S'Base'Last is the base range of the type. Using an attribute_definition_clause, one cannot specify any subtype-specific attributes for the subtype denoted by S'Base (the base subtype).

0.c The attribute S'Range is now allowed for scalar subtypes.

0.d The attributes S'Min and S'Max are now defined, and made available for all scalar types.

0.e The attributes S'Succ, S'Pred, S'Image, S'Value, and S'Width are now defined for real types as well as discrete types.

0.f Wide_String versions of S'Image and S'Value are defined. These are called S'Wide_Image and S'Wide_Value to avoid introducing ambiguities involving uses of these attributes with string literals.

Wording Changes From Ada 83

0.g We now use the syntactic category range_attribute_reference since it is now syntactically distinguished from other attribute references.

The definition of S'Base has been moved here from 3.3.3 since it now applies only to scalar types. 0.h

More explicit rules are provided for nongraphic characters. 0.i

3.5.1 Enumeration Types

Wording Changes From Ada 83

The syntax rule for defining_character_literal is new. It is used for the defining occurrence of a character_literal, analogously to defining_identifier. Usage occurrences use the name or selector_name syntactic categories. 0.a

We emphasize the fact that an enumeration literal denotes a function, which is called to produce a value. 0.b

3.5.2 Character Types

Inconsistencies With Ada 83

The declaration of Wide_Character in package Standard hides use-visible declarations with the same defining identifier. In the unlikely event that an Ada 83 program had depended on such a use-visible declaration, and the program remains legal after the substitution of Standard.Wide_Character, the meaning of the program will be different. 0.a

Incompatibilities With Ada 83

The presence of Wide_Character in package Standard means that an expression such as 0.b

```
    'a' = 'b'
```
0.c

is ambiguous in Ada 9X, whereas in Ada 83 both literals could be resolved to be of type Character. 0.d

The change in visibility rules (see 4.2) for character literals means that additional qualification might be necessary to resolve expressions involving overloaded subprograms and character literals. 0.e

Extensions to Ada 83

The type Character has been extended to have 256 positions, and the type Wide_Character has been added. Note that this change was already approved by the ARG for Ada 83 conforming compilers. 0.f

The rules for referencing character literals are changed (see 4.2), so that the declaration of the character type need not be directly visible to use its literals, similar to **null** and string literals. Context is used to resolve their type. 0.g

3.5.3 Boolean Types

3.5.4 Integer Types

Extensions to Ada 83

An implementation is allowed to support any number of distinct base ranges for integer types, even if fewer integer types are explicitly declared in Standard. 0.a

Modular (unsigned, wrap-around) types are new. 0.b

Wording Changes From Ada 83

Ada 83's integer types are now called "signed" integer types, to contrast them with "modular" integer types. 0.c

Standard.Integer, Standard.Long_Integer, etc., denote constrained subtypes of predefined integer types, consistent with the Ada 9X model that only subtypes have names. 0.d

We now impose minimum requirements on the base range of Integer and Long_Integer. 0.e

We no longer explain integer type definition in terms of an equivalence to a normal type derivation, except to say that all integer types are by definition implicitly derived from *root_integer*. This is for various reasons. 0.f

First of all, the equivalence with a type derivation and a subtype declaration was not perfect, and was the source of various AIs (for example, is the conversion of the bounds static? Is a numeric type a derived type with respect to other rules of the language?) 0.g

0.h Secondly, we don't want to require that every integer size supported shall have a corresponding named type in Standard. Adding named types to Standard creates nonportabilities.

0.i Thirdly, we don't want the set of types that match a formal derived type "type T is new Integer;" to depend on the particular underlying integer representation chosen to implement a given user-defined integer type. Hence, we would have needed anonymous integer types as parent types for the implicit derivation anyway. We have simply chosen to identify only one anonymous integer type — *root_integer*, and stated that every integer type is derived from it.

0.j Finally, the ''fiction'' that there were distinct preexisting predefined types for every supported representation breaks down for fixed point with arbitrary smalls, and was never exploited for enumeration types, array types, etc. Hence, there seems little benefit to pushing an explicit equivalence between integer type definition and normal type derivation.

3.5.5 Operations of Discrete Types

Extensions to Ada 83

0.a The attributes S'Succ, S'Pred, S'Width, S'Image, and S'Value have been generalized to apply to real types as well (see 3.5, ''Scalar Types'').

3.5.6 Real Types

Wording Changes From Ada 83

0.a The syntax rule for real_type_definition is modified to use the new syntactic categories floating_point_definition and fixed_point_definition, instead of floating_point_constraint and fixed_point_constraint, because the semantics of a type definition are significantly different than the semantics of a constraint.

0.b All discussion of model numbers, safe ranges, and machine numbers is moved to 3.5.7, 3.5.8, and G.2. Values of a fixed point type are now described as being multiples of the *small* of the fixed point type, and we have no need for model numbers, safe ranges, etc. for fixed point types.

3.5.7 Floating Point Types

Inconsistencies With Ada 83

0.a No Range_Checks, only Overflow_Checks, are performed on variables (or parameters) of an unconstrained floating point subtype. This is upward compatible for programs that do not raise Constraint_Error. For those that do raise Constraint_Error, it is possible that the exception will be raised at a later point, or not at all, if extended range floating point registers are used to hold the value of the variable (or parameter).

Wording Changes From Ada 83

0.b The syntax rules for floating_point_constraint and floating_accuracy_definition are removed. The syntax rules for floating_point_definition and real_range_specification are new.

0.c A syntax rule for digits_constraint is given in 3.5.9, ''Fixed Point Types''. In J.3 we indicate that a digits_constraint may be applied to a floating point subtype_mark as well (to be compatible with Ada 83's floating_point_constraint).

0.d Discussion of model numbers is postponed to 3.5.8 and G.2. The concept of safe numbers has been replaced by the concept of the safe range of values. The bounds of the safe range are given by T'Safe_First .. T'Safe_Last, rather than -T'Safe_Large .. T'Safe_Large, since on some machines the safe range is not perfectly symmetric. The concept of machine numbers is new, and is relevant to the definition of Succ and Pred for floating point numbers.

3.5.8 Operations of Floating Point Types

3.5.9 Fixed Point Types

Inconsistencies With Ada 83

0.a In Ada 9X, S'Small always equals S'Base'Small, so if an implementation chooses a *small* for a fixed point type smaller than required by the *delta*, the value of S'Small in Ada 9X might not be the same as it was in Ada 83.

Extensions to Ada 83

Decimal fixed point types are new, though their capabilities are essentially similar to that available in Ada 83 with a fixed point type whose *small* equals its *delta* equals a power of 10. However, in the Information Systems Annex, additional requirements are placed on the support of decimal fixed point types (e.g. a minimum of 18 digits of precision). 0.b

Wording Changes From Ada 83

The syntax rules for fixed_point_constraint and fixed_accuracy_definition are removed. The syntax rule for fixed_point_definition is new. A syntax rule for delta_constraint is included in the Obsolescent features (to be compatible with Ada 83's fixed_point_constraint). 0.c

3.5.10 Operations of Fixed Point Types

3.6 Array Types

Extensions to Ada 83

The syntax rule for component_definition is modified to allow the reserved word **aliased**. 0.a

The syntax rules for unconstrained_array_definition and constrained_array_definition are modified to use component_definition (instead of *component*_subtype_indication). The effect of this change is to allow the reserved word **aliased** before the component subtype_indication. 0.b

A range in a discrete_subtype_definition may use arbitrary universal expressions for each bound (e.g. –1 .. 3+5), rather than strictly "implicitly convertible" operands. The subtype defined will still be a subtype of Integer. 0.c

Wording Changes From Ada 83

We introduce a new syntactic category, discrete_subtype_definition, as distinct from discrete_range. These two constructs have the same syntax, but their semantics are quite different (one defines a subtype, with a preference for Integer subtypes, while the other just selects a subrange of an existing subtype). We use this new syntactic category in **for** loops and entry families. 0.d

The syntax for index_constraint and discrete_range have been moved to their own subclause, since they are no longer used here. 0.e

The syntax rule for component_definition (formerly component_subtype_definition) is moved here from RM83-3.7. 0.f

3.6.1 Index Constraints and Discrete Ranges

Extensions to Ada 83

We allow the declaration of a variable with a nominally unconstrained array subtype, so long as it has an initialization expression to determine its bounds. 0.a

Wording Changes From Ada 83

We have moved the syntax for index_constraint and discrete_range here since they are no longer used in constrained_array_definitions. We therefore also no longer have to describe the (special) semantics of index_constraints and discrete_ranges that appear in constrained_array_definitions. 0.b

The rules given in RM83-3.6.1(5,7-10), which define the bounds of an array object, are redundant with rules given elsewhere, and so are not repeated here. RM83-3.6.1(6), which requires that the (nominal) subtype of an array variable be constrained, no longer applies, so long as the variable is explicitly initialized. 0.c

3.6.2 Operations of Array Types

3.6.3 String Types

Inconsistencies With Ada 83

The declaration of Wide_String in Standard hides a use-visible declaration with the same defining_identifier. In rare cases, this might result in an inconsistency between Ada 83 and Ada 9X. 0.a

Incompatibilities With Ada 83

0.b Because both String and Wide_String are always directly visible, an expression like

0.c
```
    "a" < "bc"
```

0.d is now ambiguous, whereas in Ada 83 both string literals could be resolved to type String.

Extensions to Ada 83

0.e The type Wide_String is new (though it was approved by ARG for Ada 83 compilers as well).

Wording Changes From Ada 83

0.f We define the term *string type* as a natural analogy to the term *character type*.

3.7 Discriminants

Extensions to Ada 83

0.a The syntax for a discriminant_specification is modified to allow an *access discriminant*, with a type specified by an access_definition (see 3.10).

0.b Discriminants are allowed on all composite types other than array types.

0.c Discriminants may be of an access type.

Wording Changes From Ada 83

0.d Discriminant_parts are not elaborated, though an access_definition is elaborated when the discriminant is initialized.

3.7.1 Discriminant Constraints

Inconsistencies With Ada 83

0.a Dependent compatibility checks are no longer performed on subtype declaration. Instead they are deferred until object creation (see 3.3.1). This is upward compatible for a program that does not raise Constraint_Error.

Wording Changes From Ada 83

0.b Everything in RM83-3.7.2(7-12), which specifies the initial values for discriminants, is now redundant with 3.3.1, 6.4.1, 8.5.1, and 12.4. Therefore, we don't repeat it here. Since the material is largely intuitive, but nevertheless complicated to state formally, it doesn't seem worth putting it in a "NOTE."

3.7.2 Operations of Discriminated Types

Extensions to Ada 83

0.a For consistency with other attributes, we are allowing the prefix of Constrained to be a value as well as an object of a discriminated type, and also an implicit dereference. These extensions are not important capabilities, but there seems no reason to make this attribute different from other similar attributes. We are curious what most Ada 83 compilers do with F(1).X'Constrained.

0.b We now handle in a general way the cases of erroneousness identified by AI-585, where the prefix of an indexed_component or slice is discriminant-dependent, and the evaluation of the index or discrete range changes the value of a discriminant.

Wording Changes From Ada 83

0.c We have moved all discussion of erroneous use of names that denote discriminant-dependent subcomponents to this subclause. In Ada 83, it used to appear separately under assignment_statements and subprogram calls.

3.8 Record Types

Extensions to Ada 83

0.a The syntax rule for component_declaration is modified to use component_definition (instead of component_subtype_definition). The effect of this change is to allow the reserved word **aliased** before the component_subtype_definition.

0.b A short-hand is provided for defining a null record type (and a null record extension), as these will be more common for abstract root types (and derived types without additional components).

The syntax rule for record_type_definition is modified to allow the reserved words **tagged** and **limited**. Tagging is new. Limitedness is now orthogonal to privateness. In Ada 83 the syntax implied that limited private was sort of more private than private. However, limitedness really has nothing to do with privateness; limitedness simply indicates the lack of assignment capabilities, and makes perfect sense for nonprivate types such as record types. 0.c

Wording Changes From Ada 83

The syntax rules now allow representation_clauses to appear in a record_definition. This is not a language extension, because Legality Rules prevent all language-defined representation clauses from appearing there. However, an implementation-defined attribute_definition_clause could appear there. The reason for this change is to allow the rules for representation_clauses and representation pragmas to be as similar as possible. 0.d

3.8.1 Variant Parts and Discrete Choices

Extensions to Ada 83

In Ada 83, the discriminant of a variant_part is not allowed to be of a generic formal type. This restriction is removed in Ada 9X; an **others** discrete_choice is required in this case. 0.a

Wording Changes From Ada 83

The syntactic category choice is removed. The syntax rules for variant, array_aggregate, and case_statement now use discrete_choice_list or discrete_choice instead. The syntax rule for record_aggregate now defines its own syntax for named associations. 0.b

We have added the term Discrete Choice to the title since this is where they are talked about. This is analogous to the name of the subclause "Index Constraints and Discrete Ranges" in the clause on Array Types. 0.c

The rule requiring that the discriminant denote a discriminant of the type being defined seems to have been left implicit in RM83. 0.d

3.9 Tagged Types and Type Extensions

Extensions to Ada 83

Tagged types are a new concept. 0.a

3.9.1 Type Extensions

Extensions to Ada 83

Type extension is a new concept. 0.a

3.9.2 Dispatching Operations of Tagged Types

Extensions to Ada 83

The concept of dispatching operations is new. 0.a

3.9.3 Abstract Types and Subprograms

3.10 Access Types

Extensions to Ada 83

The syntax for access_type_definition is changed to support general access types (including access-to-constants) and access-to-subprograms. The syntax rules for general_access_modifier and access_definition are new. 0.a

Wording Changes From Ada 83

We use the term "storage pool" to talk about the data area from which allocation takes place. The term "collection" is no longer used. ("Collection" and "storage pool" are not the same thing because multiple unrelated access types can share the same storage pool; see 13.11 for more discussion.) 0.b

3.10.1 Incomplete Type Declarations

Extensions to Ada 83

0.a The full_type_declaration that completes an incomplete_type_declaration may have a known_discriminant_part even if the incomplete_type_declaration does not.

0.b A discriminant_constraint may be applied to an incomplete type, even if it its completion is deferred to the package body, because there is no ‘‘dependent compatibility check’’ required any more. Of course, the constraint can be specified only if a known_discriminant_part was given in the incomplete_type_declaration. As mentioned in the previous paragraph, that is no longer required even when the full type has discriminants.

Wording Changes From Ada 83

0.c Dereferences producing incomplete types were not explicitly disallowed in RM83, though AI-00039 indicated that it was not strictly necessary since troublesome cases would result in Constraint_Error at run time, since the access value would necessarily be null. However, this introduces an undesirable implementation burden, as illustrated by Example 4 of AI-00039:

0.d
```
package Pack is
    type Pri is private;
private
    type Sep;
    type Pri is access Sep;
    X : Pri;
end Pack;
```

0.e
```
package body Pack is -- Could be separately compiled!
    type Sep is ...;
    X := new Sep;
end Pack;
```

0.f
```
pragma Elaborate(Pack);
private package Pack.Child is
    I : Integer := X.all'Size; -- Legal, by AI-00039.
end Pack.Child;
```

0.g Generating code for the above example could be a serious implementation burden, since it would require all aliased objects to store size dope, and for that dope to be in the same format for all kinds of types (or some other equivalently inefficient implementation). On the contrary, most implementations allocate dope differently (or not at all) for different designated subtypes.

3.10.2 Operations of Access Types

Extensions to Ada 83

0.a We no longer make things like ’Last and ".component" (basic) operations of an access type that need to be "declared" somewhere. Instead, implicit dereference in a prefix takes care of them all. This means that there should never be a case when X.**all**’Last is legal while X’Last is not. See AI-00154.

3.11 Declarative Parts

Extensions to Ada 83

0.a The syntax for declarative_part is modified to remove the ordering restrictions of Ada 83; that is, the distinction between basic_declarative_items and later_declarative_items within declarative_parts is removed. This means that things like use_clauses and variable_declarations can be freely intermixed with things like bodies.

0.b The syntax rule for proper_body now allows a protected_body, and the rules for elaboration checks now cover calls on protected operations.

Wording Changes From Ada 83

0.c The syntax rule for later_declarative_item is removed; the syntax rule for declarative_item is new.

0.d RM83 defines ‘‘elaborated’’ and ‘‘not yet elaborated’’ for declarative_items here, and for other things in 3.1, ‘‘Declarations’’. That’s no longer necessary, since these terms are fully defined in 3.1.

0.e In RM83, all uses of declarative_part are optional (except for the one in block_statement with a **declare**) which is sort of strange, since a declarative_part can be empty, according to the syntax. That is, declarative_parts are sort of ‘‘doubly optional’’. In Ada 9X, these declarative_parts are always required (but can still be empty). To simplify description, we go further and say (see 5.6, ‘‘Block Statements’’) that a block_statement without an explicit declarative_part is equivalent to one with an empty one.

3.11.1 Completions of Declarations

Wording Changes From Ada 83

This subclause is new. It is intended to cover all kinds of completions of declarations, be they a body for a spec, a full type for an incomplete or private type, a full constant declaration for a deferred constant declaration, or a pragma Import for any kind of entity. 0.a

Chg839X;5.95

Section 4: Names and Expressions

4.1 Names

Extensions to Ada 83

Type conversions and function calls are now considered names that denote the result of the operation. In the case of a type conversion used as an actual parameter or that is of a tagged type, the type conversion is considered a variable if the operand is a variable. This simplifies the description of "parameters of the form of a type conversion" as well as better supporting an important OOP paradigm that requires the combination of a conversion from a class-wide type to some specific type followed immediately by component selection. Function calls are considered names so that a type conversion of a function call and the function call itself are treated equivalently in the grammar. A function call is considered the name of a constant, and can be used anywhere such a name is permitted. See 6.5. 0.a

Type conversions of a tagged type are permitted anywhere their operand is permitted. That is, if the operand is a variable, then the type conversion can appear on the left-hand side of an assignment_statement. If the operand is an object, then the type conversion can appear in an object renaming or as a prefix. See 4.6. 0.b

Wording Changes From Ada 83

Everything of the general syntactic form name(...) is now syntactically a name. In any realistic parser, this would be a necessity since distinguishing among the various name(...) constructs inevitably requires name resolution. In cases where the construct yields a value rather than an object, the name denotes the value rather than an object. Names already denote values in Ada 83 with named numbers, components of the result of a function call, etc. This is partly just a wording change, and partly an extension of functionality (see Extensions heading above). 0.c

The syntax rule for direct_name is new. It is used in places where direct visibility is required. It's kind of like Ada 83's simple_name, but simple_name applied to both direct visibility and visibility by selection, and furthermore, it didn't work right for operator_symbols. The syntax rule for simple_name is removed, since its use is covered by a combination of direct_name and selector_name. The syntactic categories direct_name and selector_name are similar; it's mainly the visibility rules that distinguish the two. The introduction of direct_name requires the insertion of one new explicit textual rule: to forbid statement_identifiers from being operator_symbols. This is the only case where the explicit rule is needed, because this is the only case where the declaration of the entity is implicit. For example, there is no need to syntactically forbid (say) ''X: "Rem";'', because it is impossible to declare a type whose name is an operator_symbol in the first place. 0.d

The syntax rules for explicit_dereference and implicit_dereference are new; this makes other rules simpler, since dereferencing an access value has substantially different semantics from selected_components. We also use name instead of prefix in the explicit_dereference rule since that seems clearer. Note that these rules rely on the fact that function calls are now names, so we don't need to use prefix to allow functions calls in front of .**all**. 0.e

We no longer use the term *appropriate for a type* since we now describe the semantics of a prefix in terms of implicit dereference. 0.f

4.1.1 Indexed Components

4.1.2 Slices

4.1.3 Selected Components

Extensions to Ada 83

We now allow an expanded name to use a prefix that denotes a rename of a package, even if the selector is for an entity local to the body or private part of the package, so long as the entity is visible at the place of the reference. This eliminates a preexisting anomaly where references in a package body may refer to declarations of its visible part but not those of its private part or body when the prefix is a rename of the package. 0.a

Wording Changes From Ada 83

The syntax rule for selector_name is new. It is used in places where visibility, but not necessarily direct visibility, is required. See 4.1, ''Names'' for more information. 0.b

The description of dereferencing an access type has been moved to 4.1, ''Names''; name.**all** is no longer considered a selected_component. 0.c

0.d The rules have been restated to be consistent with our new terminology, to accommodate class-wide types, etc.

4.1.4 Attributes

Extensions to Ada 83

0.a We now uniformly treat X'Range as X'First..X'Last, allowing its use with scalar subtypes.

0.b We allow any integer type in the *static_*expression of an attribute designator, not just a value of *universal_integer*. The preference rules ensure upward compatibility.

Wording Changes From Ada 83

0.c We use the syntactic category attribute_reference rather than simply "attribute" to avoid confusing the name of something with the thing itself.

0.d The syntax rule for attribute_reference now uses identifier instead of simple_name, because attribute identifiers are not required to follow the normal visibility rules.

0.e We now separate attribute_reference from range_attribute_reference, and enumerate the reserved words that are legal attribute or range attribute designators. We do this because identifier no longer includes reserved words.

0.f The Ada 9X name resolution rules are a bit more explicit than in Ada 83. The Ada 83 rule said that the "meaning of the prefix of an attribute must be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute." That isn't quite right since the meaning even in Ada 83 embodies whether or not the prefix is interpreted as a parameterless function call, and in Ada 9X, it also embodies whether or not the prefix is interpreted as an implicit_dereference. So the attribute designator does make a difference — just not much.

0.g Note however that if the attribute designator is Access, it makes a big difference in the interpretation of the prefix (see 3.10.2).

4.2 Literals

Incompatibilities With Ada 83

0.a Because character_literals are now treated like other literals, in that they are resolved using context rather than depending on direct visibility, additional qualification might be necessary when passing a character_literal to an overloaded subprogram.

Extensions to Ada 83

0.b Character_literals are now treated analogously to **null** and string_literals, in that they are resolved using context, rather than their content; the declaration of the corresponding defining_character_literal need not be directly visible.

Wording Changes From Ada 83

0.c Name Resolution rules for enumeration literals that are not character_literals are not included anymore, since they are neither syntactically nor semantically "literals" but are rather names of parameterless functions.

4.3 Aggregates

Extensions to Ada 83

0.a We now allow extension_aggregates.

Wording Changes From Ada 83

0.b We have adopted new wording for expressing the rule that the type of an aggregate shall be determinable from the outside, though using the fact that it is nonlimited record (extension) or array.

0.c An aggregate now creates an anonymous object. This is necessary so that controlled types will work (see 7.6).

4.3.1 Record Aggregates

Extensions to Ada 83

0.a Null record aggregates may now be specified, via "(**null record**)". However, this syntax is more useful for null record extensions in extension aggregates.

Wording Changes From Ada 83

Various AIs have been incorporated (AI-189, AI-244, and AI-309). In particular, Ada 83 did not explicitly disallow extra values in a record aggregate. Now we do. 0.b

4.3.2 Extension Aggregates

Extensions to Ada 83

The extension aggregate syntax is new. 0.a

4.3.3 Array Aggregates

Extensions to Ada 83

We now allow "named with others" aggregates in all contexts where there is an applicable index constraint, effectively eliminating what was RM83-4.3.2(6). Sliding never occurs on an aggregate with others, because its bounds come from the applicable index constraint, and therefore already match the bounds of the target. 0.a

The legality of an **others** choice is no longer affected by the staticness of the applicable index constraint. This substantially simplifies several rules, while being slightly more flexible for the user. It obviates the rulings of AI-244 and AI-310, while taking advantage of the dynamic nature of the "extra values" check required by AI-309. 0.b

Named array aggregates are permitted even if the index type is descended from a formal scalar type. See 4.9 and AI-00190. 0.c

Wording Changes From Ada 83

We now separate named and positional array aggregate syntax, since, unlike other aggregates, named and positional associations cannot be mixed in array aggregates (except that an **others** choice is allowed in a positional array aggregate). 0.d

We have also reorganized the presentation to handle multidimensional and one-dimensional aggregates more uniformly, and to incorporate the rulings of AI-19, AI-309, etc. 0.e

4.4 Expressions

Extensions to Ada 83

In Ada 83, **out** parameters and their nondiscriminant subcomponents are not allowed as primaries. These restrictions are eliminated in Ada 9X. 0.a

In various contexts throughout the language where Ada 83 syntax rules had simple_expression, the corresponding Ada 9X syntax rule has expression instead. This reflects the inclusion of modular integer types, which makes the logical operators "**and**", "**or**", and "**xor**" more useful in expressions of an integer type. Requiring parentheses to use these operators in such contexts seemed unnecessary and potentially confusing. Note that the bounds of a range still have to be specified by simple_expressions, since otherwise expressions involving membership tests might be ambiguous. Essentially, the operation ".." is of higher precedence than the logical operators, and hence uses of logical operators still have to be parenthesized when used in a bound of a range. 0.b

4.5 Operators and Expression Evaluation

Wording Changes From Ada 83

We don’t give a detailed definition of precedence, since it is all implicit in the syntax rules anyway. 0.a

The permission to reassociate is moved here from RM83-11.6(5), so it is closer to the rules defining operator association. 0.b

4.5.1 Logical Operators and Short-circuit Control Forms

4.5.2 Relational Operators and Membership Tests

Extensions to Ada 83

0.a Membership tests can be used to test the tag of a class-wide value.

0.b Predefined equality for a composite type is defined in terms of the primitive equals operator for tagged components or the parent part.

Wording Changes From Ada 83

0.c The term ‘‘membership test’’ refers to the relation "X in S" rather to simply the reserved word **in** or **not in**.

0.d We use the term ‘‘equality operator’’ to refer to both the = (equals) and /= (not equals) operators. Ada 83 referred to = as *the* equality operator, and /= as the inequality operator. The new wording is more consistent with the ISO 10646 name for "=" (equals sign) and provides a category similar to ‘‘ordering operator’’ to refer to both = and /=.

0.e We have changed the term ‘‘catenate’’ to ‘‘concatenate’’.

4.5.3 Binary Adding Operators

Inconsistencies With Ada 83

0.a The lower bound of the result of concatenation, for a type whose first subtype is constrained, is now that of the index subtype. This is inconsistent with Ada 83, but generally only for Ada 83 programs that raise Constraint_Error. For example, the concatenation operator in

0.b
```
X : array(1..10) of Integer;
begin
X := X(6..10) & X(1..5);
```

0.c would raise Constraint_Error in Ada 83 (because the bounds of the result of the concatenation would be 6..15, which is outside of 1..10), but would succeed and swap the halves of X (as expected) in Ada 9X.

Extensions to Ada 83

0.d Concatenation is now useful for array types whose first subtype is constrained. When the result type of a concatenation is such an array type, Constraint_Error is avoided by effectively first sliding the left operand (if nonnull) so that its lower bound is that of the index subtype.

4.5.4 Unary Adding Operators

4.5.5 Multiplying Operators

Extensions to Ada 83

0.a Explicit conversion of the result of multiplying or dividing two fixed point numbers is no longer required, provided the context uniquely determines some specific fixed point result type. This is to improve support for decimal fixed point, where requiring explicit conversion on every fixed-fixed multiply or divide was felt to be inappropriate.

0.b The type *universal_fixed* is covered by *universal_real*, so real literals and fixed point operands may be multiplied or divided directly, without any explicit conversions required.

Wording Changes From Ada 83

0.c We have used the normal syntax for function definition rather than a tabular format.

4.5.6 Highest Precedence Operators

Wording Changes From Ada 83

0.a We now show the specification for "**" for integer types with a parameter subtype of Natural rather than Integer for the exponent. This reflects the fact that Constraint_Error is raised if a negative value is provided for the exponent.

4.6 Type Conversions

Incompatibilities With Ada 83

A character_literal is not allowed as the operand of a type_conversion, since there are now two character types in package Standard. 0.a

The component subtypes have to statically match in an array conversion, rather than being checked for matching constraints at run time. 0.b

Because sliding of array bounds is now provided for operations where it was not in Ada 83, programs that used to raise Constraint_Error might now continue executing and produce a reasonable result. This is likely to fix more bugs than it creates. 0.c

Extensions to Ada 83

A type_conversion is considered the name of an object in certain circumstances (such a type_conversion is called a view conversion). In particular, as in Ada 83, a type_conversion can appear as an **in out** or **out** actual parameter. In addition, if the target type is tagged and the operand is the name of an object, then so is the type_conversion, and it can be used as the prefix to a selected_component, in an object_renaming_declaration, etc. 0.d

We no longer require type-mark conformance between a parameter of the form of a type conversion, and the corresponding formal parameter. This had caused some problems for inherited subprograms (since there isn't really a type-mark for converted formals), as well as for renamings, formal subprograms, etc. See AI-245, AI-318, AI-547. 0.e

We now specify ‘‘deterministic’’ rounding from real to integer types when the value of the operand is exactly between two integers (rounding is away from zero in this case). 0.f

‘‘Sliding’’ of array bounds (which is part of conversion to an array subtype) is performed in more cases in Ada 9X than in Ada 83. Sliding is not performed on the operand of a membership test, nor on the operand of a qualified_expression. It wouldn't make sense on a membership test, and we wish to retain a connection between subtype membership and subtype qualification. In general, a subtype membership test returns True if and only if a corresponding subtype qualification succeeds without raising an exception. Other operations that take arrays perform sliding. 0.g

Wording Changes From Ada 83

We no longer explicitly list the kinds of things that are not allowed as the operand of a type_conversion, except in a NOTE. 0.h

The rules in this clause subsume the rules for "parameters of the form of a type conversion," and have been generalized to cover the use of a type conversion as a name. 0.i

4.7 Qualified Expressions

4.8 Allocators

Incompatibilities With Ada 83

The subtype_indication of an uninitialized allocator may not have an explicit constraint if the designated type is an access type. In Ada 83, this was permitted even though the constraint had no affect on the subtype of the created object. 0.a

Extensions to Ada 83

Allocators creating objects of type *T* are now overloaded on access types designating *T*'Class and all class-wide types that cover *T*. 0.b

Implicit array subtype conversion (sliding) is now performed as part of an initialized allocator. 0.c

Wording Changes From Ada 83

We have used a new organization, inspired by the ACID document, that makes it clearer what is the subtype of the created object, and what subtype conversions take place. 0.d

Discussion of storage management issues, such as garbage collection and the raising of Storage_Error, has been moved to 13.11, ‘‘Storage Management’’. 0.e

4.9 Static Expressions and Static Subtypes

Extensions to Ada 83

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

0.b • Membership tests and short-circuit control forms may appear in a static expression.

0.c • The bounds and length of statically constrained array objects or subtypes are static.

0.d • The Range attribute of a statically constrained array subtype or object gives a static range.

0.e • A type_conversion is static if the subtype_mark denotes a static scalar subtype and the operand is a static expression.

0.f • All numeric literals are now static, even if the expected type is a formal scalar type. This is useful in case_statements and variant_parts, which both now allow a value of a formal scalar type to control the selection, to ease conversion of a package into a generic package. Similarly, named array aggregates are also permitted for array types with an index type that is a formal scalar type.

0.g The rules for the evaluation of static expressions are revised to require exact evaluation at compile time, and force a machine number result when crossing from the static realm to the dynamic realm, to enhance portability and predictability. Exact evaluation is not required for descendants of a formal scalar type, to simplify generic code sharing and to avoid generic contract model problems.

0.h Static expressions are legal even if an intermediate in the expression goes outside the base range of the type. Therefore, the following will succeed in Ada 9X, whereas it might raise an exception in Ada 83:

0.i
```
type Short_Int is range -32_768 .. 32_767;
I : Short_Int := -32_768;
```

0.j This might raise an exception in Ada 83 because "32_768" is out of range, even though "–32_768" is not. In Ada 9X, this will always succeed.

0.k Certain expressions involving string operations (in particular concatenation and membership tests) are considered static in Ada 9X.

0.l The reason for this change is to simplify the rule requiring compile-time-known string expressions as the link name in an interfacing pragma, and to simplify the preelaborability rules.

Incompatibilities With Ada 83

0.m An Ada 83 program that uses an out-of-range static value is illegal in Ada 9X, unless the expression is part of a larger static expression, or the expression is not evaluated due to being on the right-hand side of a short-circuit control form.

Wording Changes From Ada 83

0.n This clause (and 4.5.5, ‘‘Multiplying Operators’’) subsumes the RM83 section on Universal Expressions.

0.o The existence of static string expressions necessitated changing the definition of static subtype to include string subtypes. Most occurrences of "static subtype" have been changed to "static scalar subtype", in order to preserve the effect of the Ada 83 rules. This has the added benefit of clarifying the difference between "static subtype" and "statically constrained subtype", which has been a source of confusion. In cases where we allow static string subtypes, we explicitly use phrases like "static string subtype" or "static (scalar or string) subtype", in order to clarify the meaning for those who have gotten used to the Ada 83 terminology.

0.p In Ada 83, an expression was considered nonstatic if it raised an exception. Thus, for example:

0.q
```
Bad: constant := 1/0; -- Illegal!
```

0.r was illegal because 1/0 was not static. In Ada 9X, the above example is still illegal, but for a different reason: 1/0 is static, but there’s a separate rule forbidding the exception raising.

4.9.1 Statically Matching Constraints and Subtypes

Wording Changes From Ada 83

0.a This subclause is new to Ada 9X.

Section 5: Statements

Wording Changes From Ada 83

The description of return_statements has been moved to 6.5, ‘‘Return Statements’’, so that it is closer to the description of subprograms. 0.a

5.1 Simple and Compound Statements - Sequences of Statements

Extensions to Ada 83

The requeue_statement is new. 0.a

Wording Changes From Ada 83

We define the syntactic category statement_identifier to simplify the description. It is used for labels, loop names, and block names. We define the entity associated with the implicit declarations of statement names. 0.b

Completion includes completion caused by a transfer of control, although RM83-5.1(6) did not take this view. 0.c

5.2 Assignment Statements

Extensions to Ada 83

We now allow user-defined finalization and value adjustment actions as part of assignment_statements (see 7.6, ‘‘User-Defined Assignment and Finalization’’). 0.a

Wording Changes From Ada 83

The special case of array assignment is subsumed by the concept of a subtype conversion, which is applied for all kinds of types, not just arrays. For arrays it provides ‘‘sliding.’’ For numeric types it provides conversion of a value of a universal type to the specific type of the target. For other types, it generally has no run-time effect, other than a constraint check. 0.b

We now cover in a general way in 3.7.2 the erroneous execution possible due to changing the value of a discriminant when the variable in an assignment_statement is a subcomponent that depends on discriminants. 0.c

5.3 If Statements

5.4 Case Statements

Extensions to Ada 83

In Ada 83, the expression in a case_statement is not allowed to be of a generic formal type. This restriction is removed in Ada 9X; an **others** discrete_choice is required instead. 0.a

In Ada 9X, a function call is the name of an object; this was not true in Ada 83 (see 4.1, ‘‘Names’’). This change makes the following case_statement legal: 0.b

0.c
```
subtype S is Integer range 1..2;
function F return S;
case F is
   when 1 => ...;
   when 2 => ...;
   -- No others needed.
end case;
```

Note that the result subtype given in a function renaming_declaration is ignored; for a case_statement whose expression calls a such a function, the full coverage rules are checked using the result subtype of the original function. Note that predefined operators such as "+" have an unconstrained result subtype (see 4.5.1). Note that generic formal functions do not have static result subtypes. Note that the result subtype of an inherited subprogram need not correspond to any namable subtype; there is still a perfectly good result subtype, though. 0.d

Wording Changes From Ada 83

Ada 83 forgot to say what happens for ‘‘legally’’ out-of-bounds values. 0.e

0.f We take advantage of rules and terms (e.g. *cover a value*) defined for discrete_choices and discrete_choice_lists in 3.8.1, ‘‘Variant Parts and Discrete Choices’’.

0.g In the Name Resolution Rule for the case expression, we no longer need RM83-5.4(3)’s ‘‘which must be determinable independently of the context in which the expression occurs, but using the fact that the expression must be of a discrete type,’’ because the expression is now a complete context. See 8.6, ‘‘The Context of Overload Resolution’’.

0.h Since type_conversions are now defined as names, their coverage rule is now covered under the general rule for names, rather than being separated out along with qualified_expressions.

5.5 Loop Statements

Wording Changes From Ada 83

0.a The constant-ness of loop parameters is specified in 3.3, ‘‘Objects and Named Numbers’’.

5.6 Block Statements

Wording Changes From Ada 83

0.a The syntax rule for block_statement now uses the syntactic category handled_sequence_of_statements.

5.7 Exit Statements

5.8 Goto Statements

Section 6: Subprograms

6.1 Subprogram Declarations

Extensions to Ada 83

The syntax for abstract_subprogram_declaration is added. The syntax for parameter_specification is revised to allow for access parameters (see 3.10) 0.a

Program units that are library units may have a parent_unit_name to indicate the parent of a child (see Section 10). 0.b

Wording Changes From Ada 83

We have incorporated the rules from RM83-6.5, ‘‘Function Subprograms’’ here and in 6.3, ‘‘Subprogram Bodies’’ 0.c

We have incorporated the definitions of RM83-6.6, ‘‘Parameter and Result Type Profile - Overloading of Subprograms’’ here. 0.d

The syntax rule for defining_operator_symbol is new. It is used for the defining occurrence of an operator_symbol, analogously to defining_identifier. Usage occurrences use the direct_name or selector_name syntactic categories. The syntax rules for defining_designator and defining_program_unit_name are new. 0.e

6.2 Formal Parameter Modes

Extensions to Ada 83

The value of an **out** parameter may be read. An **out** parameter is treated like a declared variable without an explicit initial expression. 0.a

Wording Changes From Ada 83

Discussion of copy-in for parts of out parameters is now covered in 6.4.1, ‘‘Parameter Associations’’. 0.b

The concept of a by-reference type is new to Ada 9X. 0.c

We now cover in a general way in 3.7.2 the rule regarding erroneous execution when a discriminant is changed and one of the parameters depends on the discriminant. 0.d

6.3 Subprogram Bodies

Extensions to Ada 83

A renaming_declaration may be used instead of a subprogram_body. 0.a

Wording Changes From Ada 83

The syntax rule for subprogram_body now uses the syntactic category handled_sequence_of_statements. 0.b

The declarative_part of a subprogram_body is now required; that doesn’t make any real difference, because a declarative_part can be empty. 0.c

We have incorporated some rules from RM83-6.5 here. 0.d

RM83 forgot to restrict the definition of elaboration of a subprogram_body to non-generics. 0.e

6.3.1 Conformance Rules

Extensions to Ada 83

The rules for full conformance are relaxed — they are now based on the structure of constructs, rather than the sequence of lexical elements. This implies, for example, that "(X, Y: T)" conforms fully with "(X: T; Y: T)", and "(X: T)" conforms fully with "(X: **in** T)". 0.a

6.3.2 Inline Expansion of Subprograms

Extensions to Ada 83

0.a A pragma Inline is allowed inside a subprogram_body if there is no corresponding subprogram_declaration. This is for uniformity with other program unit pragmas.

6.4 Subprogram Calls

Wording Changes From Ada 83

0.a We have gotten rid of parameters ‘‘of the form of a type conversion’’ (see RM83-6.4.1(3)). The new view semantics of type_conversions allows us to use normal type_conversions instead.

0.b We have moved wording about run-time semantics of parameter associations to 6.4.1.

0.c We have moved wording about raising Program_Error for a function that falls off the end to here from RM83-6.5.

6.4.1 Parameter Associations

Extensions to Ada 83

0.a In Ada 9X, a program can rely on the fact that passing an object as an **out** parameter does not ‘‘de-initialize’’ any parts of the object whose subtypes have implicit initial values. (This generalizes the RM83 rule that required copy-in for parts that were discriminants or of an access type.)

Wording Changes From Ada 83

0.b We have eliminated the subclause on Default Parameters, as it is subsumed by earlier clauses and subclauses.

6.5 Return Statements

Incompatibilities With Ada 83

0.a In Ada 9X, if the result type of a function has a part that is a task, then an attempt to return a local variable will raise Program_Error. In Ada 83, if a function returns a local variable containing a task, execution is erroneous according to AI-00867. However, there are other situations where functions that return tasks (or that return a variant record only one of whose variants includes a task) are correct in Ada 83 but will raise Program_Error according to the new rules.

0.b The rule change was made because there will be more types (protected types, limited controlled types) in Ada 9X for which it will be meaningless to return a local variable, and making all of these erroneous is unacceptable. The current rule was felt to be the simplest that kept upward incompatibilities to situations involving returning tasks, which are quite rare.

Wording Changes From Ada 83

0.c This clause has been moved here from chapter 5, since it has mainly to do with subprograms.

0.d A function now creates an anonymous object. This is necessary so that controlled types will work.

0.e We have clarified that a return_statement applies to a callable construct, not to a callable entity.

0.f There is no need to mention generics in the rules about where a return_statement can appear and what it applies to; the phrase ‘‘body of a subprogram or generic subprogram’’ is syntactic, and refers exactly to ‘‘subprogram_body’’.

6.6 Overloading of Operators

Extensions to Ada 83

0.a Explicit declarations of "=" are now permitted for any combination of parameter and result types.

0.b Explicit declarations of "/=" are now permitted, so long as the result type is not Boolean.

Section 7: Packages

7.1 Package Specifications and Declarations

Incompatibilities With Ada 83

0.a In Ada 83, a library package is allowed to have a body even if it doesn't need one. In Ada 9X, a library package body is either required or forbidden — never optional. The workaround is to add **pragma** Elaborate_Body, or something else requiring a body, to each library package that has a body that isn't otherwise required.

Wording Changes From Ada 83

0.b We have moved the syntax into this clause and the next clause from RM83-7.1, ''Package Structure'', which we have removed.

0.c RM83 was unclear on the rules about when a package requires a body. For example, RM83-7.1(4) and RM83-7.1(8) clearly forgot about the case of an incomplete type declared in a package_declaration but completed in the body. In addition, RM83 forgot to make this rule apply to a generic package. We have corrected these rules. Finally, since we now allow a pragma Import for any explicit declaration, the completion rules need to take this into account as well.

7.2 Package Bodies

Wording Changes From Ada 83

0.a The syntax rule for package_body now uses the syntactic category handled_sequence_of_statements.

0.b The declarative_part of a package_body is now required; that doesn't make any real difference, since a declarative_part can be empty.

0.c RM83 seems to have forgotten to say that a package_body can't stand alone, without a previous declaration. We state that rule here.

0.d RM83 forgot to restrict the definition of elaboration of package_bodies to nongeneric ones. We have corrected that omission.

0.e The rule about implicit bodies (from RM83-9.3(5)) is moved here, since it is more generally applicable.

7.3 Private Types and Private Extensions

Extensions to Ada 83

0.a The syntax for a private_type_declaration is augmented to allow the reserved word **tagged**.

0.b In Ada 83, a private type without discriminants cannot be completed with a type with discriminants. Ada 9X allows the full view to have discriminants, so long as they have defaults (that is, so long as the first subtype is definite). This change is made for uniformity with generics, and because the rule as stated is simpler and easier to remember than the Ada 83 rule. In the original version of Ada 83, the same restriction applied to generic formal private types. However, the restriction was removed by the ARG for generics. In order to maintain the ''generic contract/private type contract analogy'' discussed above, we have to apply the same rule to package-private types. Note that a private untagged type without discriminants can be completed with a tagged type with discriminants only if the full view is constrained, because discriminants of tagged types cannot have defaults.

Wording Changes From Ada 83

0.c RM83-7.4.1(4), ''Within the specification of the package that declares a private type and before the end of the corresponding full type declaration, a restriction applies....'', is subsumed (and corrected) by the rule that a type shall be completely defined before it is frozen, and the rule that the parent type of a derived type declaration shall be completely defined, unless the derived type is a private extension.

7.3.1 Private Operations

Wording Changes From Ada 83

0.a The phrase in RM83-7.4.2(7), ''...after the full type declaration'', doesn't work in the presence of child units, so we define that rule in terms of visibility.

0.b The definition of the Constrained attribute for private types has been moved to ‘‘Obsolescent Features.’’ (The Constrained attribute of an object has not been moved there.)

7.4 Deferred Constants

Extensions to Ada 83

0.a In Ada 83, a deferred constant is required to be of a private type declared in the same visible part. This restriction is removed for Ada 9X; deferred constants can be of any type.

0.b In Ada 83, a deferred constant declaration was not permitted to include a constraint, nor the reserved word **aliased**.

0.c In Ada 83, the rules required conformance of type marks; here we require static matching of subtypes if the deferred constant is constrained.

0.d A deferred constant declaration can be completed with a pragma Import. Such a deferred constant declaration need not be within a package_specification.

0.e The rules for too-early uses of deferred constants are modified in Ada 9X to allow more cases, and catch all errors at compile time. This change is necessary in order to allow deferred constants of a tagged type without violating the principle that for a dispatching call, there is always an implementation to dispatch to. It has the beneficial side-effect of catching some Ada-83-erroneous programs at compile time. The new rule fits in well with the new freezing-point rules. Furthermore, we are trying to convert undefined-value problems into bounded errors, and we were having trouble for the case of deferred constants. Furthermore, uninitialized deferred constants cause trouble for the shared variable / tasking rules, since they are really variable, even though they purport to be constant. In Ada 9X, they cannot be touched until they become constant.

0.f Note that we do not consider this change to be an upward incompatibility, because it merely changes an erroneous execution in Ada 83 into a compile-time error.

0.g The Ada 83 semantics are unclear in the case where the full view turns out to be an access type. It is a goal of the language design to prevent uninitialized access objects. One wonders if the implementation is required to initialize the deferred constant to null, and then initialize it (again!) to its real value. In Ada 9X, the problem goes away.

Wording Changes From Ada 83

0.h Since deferred constants can now be of a nonprivate type, we have made this a stand-alone clause, rather than a subclause of 7.3, ‘‘Private Types and Private Extensions’’.

0.i Deferred constant declarations used to have their own syntax, but now they are simply a special case of object_ declarations.

7.5 Limited Types

Extensions to Ada 83

0.a The restrictions in RM83-7.4.4(4), which disallowed **out** parameters of limited types in certain cases, are removed.

Wording Changes From Ada 83

0.b Since limitedness and privateness are orthogonal in Ada 9X (and to some extent in Ada 83), this is now its own clause rather than being a subclause of 7.3, ‘‘Private Types and Private Extensions’’.

7.6 User-Defined Assignment and Finalization

Extensions to Ada 83

0.a Controlled types and user-defined finalization are new to Ada 9X. (Ada 83 had finalization semantics only for masters of tasks.)

7.6.1 Completion and Finalization

Wording Changes From Ada 83

0.a Finalization depends on the concepts of completion and leaving, and on the concept of a master. Therefore, we have moved the definitions of these concepts here, from where they used to be in Section 9. These concepts also needed to be generalized somewhat. Task waiting is closely related to user-defined finalization; the rules here refer to the task-waiting rules of Section 9.

Section 8: Visibility Rules

Wording Changes From Ada 83

We no longer define the term ‘‘basic operation;’’ thus we no longer have to worry about the visibility of them. Since they were essentially always visible in Ada 83, this change has no effect. The reason for this change is that the definition in Ada 83 was confusing, and not quite correct, and we found it difficult to fix. For example, one wonders why an if_statement was not a basic operation of type Boolean. For another example, one wonders what it meant for a basic operation to be ‘‘inherent in’’ something. Finally, this fixes the problem addressed by AI-00027/07. 0.a

8.1 Declarative Region

Wording Changes From Ada 83

It was necessary to extend Ada 83’s definition of declarative region to take the following Ada 9X features into account: 0.a

- Child library units. 0.b
- Derived types/type extensions — we need a declarative region for inherited components and also for new components. 0.c
- All the kinds of types that allow discriminants. 0.d
- Protected units. 0.e
- Entries that have bodies instead of accept statements. 0.f
- The choice_parameter_specification of an exception_handler. 0.g
- The formal parameters of access-to-subprogram types. 0.h
- Renamings-as-body. 0.i

Discriminated and access-to-subprogram type declarations need a declarative region. Enumeration type declarations cannot have one, because you don’t have to say "Color.Red" to refer to the literal Red of Color. For other type declarations, it doesn’t really matter whether or not there is an associated declarative region, so for simplicity, we give one to all types except enumeration types. 0.j

We now say that an accept_statement has its own declarative region, rather than being part of the declarative region of the entry_declaration, so that declarative regions are properly nested regions of text, so that it makes sense to talk about "inner declarative regions," and "...extends to the end of a declarative region." Inside an accept_statement, the name of one of the parameters denotes the parameter_specification of the accept_statement, not that of the entry_declaration. If the accept_statement is nested within a block_statement, these parameter_specifications can hide declarations of the block_statement. The semantics of such cases was unclear in RM83. 0.k

Note that we can’t generalize this to entry_bodies, or other bodies, because the declarative_part of a body is not supposed to contain (explicit) homographs of things in the declaration. It works for accept_statements only because an accept_statement does not have a declarative_part. 0.l

To avoid confusion, we use the term ‘‘local to’’ only informally in Ada 9X. Even RM83 used the term incorrectly (see, for example, RM83-12.3(13)). 0.m

In Ada 83, (root) library units were inside Standard; it was not clear whether the declaration or body of Standard was meant. In Ada 9X, they are children of Standard, and so occur immediately within Standard’s declarative region, but not within either the declaration or the body. (See RM83-8.6(2) and RM83-10.1.1(5).) 0.n

8.2 Scope of Declarations

Extensions to Ada 83

The fact that the immediate scope of an overloadable declaration does not include its profile is new to Ada 9X. It replaces RM83-8.3(16), which said that within a subprogram specification and within the formal part of an entry declaration or accept statement, all declarations with the same designator as the subprogram or entry were hidden from all visibility. The RM83-8.3(16) rule seemed to be overkill, and created both implementation difficulties and unnecessary semantic complexity. 0.a

Wording Changes From Ada 83

0.b We no longer need to talk about the scope of notations, identifiers, character_literals, and operator_symbols.

0.c The notion of "visible part" has been extended in Ada 9X. The syntax of task and protected units now allows private parts, thus requiring us to be able to talk about the visible part as well. It was necessary to extend the concept to subprograms and to generic units, in order for the visibility rules related to child library units to work properly. It was necessary to define the concept separately for generic formal packages, since their visible part is slightly different from that of a normal package. Extending the concept to composite types made the definition of scope slightly simpler. We define visible part for some things elsewhere, since it makes a big difference to the user for those things. For composite types and subprograms, however, the concept is used only in arcane visibility rules, so we localize it to this clause.

0.d In Ada 83, the semantics of with_clauses was described in terms of visibility. It is now described in terms of [immediate] scope.

0.e We have clarified that the following is illegal (where Q and R are library units):

0.f
```
package Q is
    I : Integer := 0;
end Q;
```

0.g
```
package R is
    package X renames Standard;
    X.Q.I := 17; -- Illegal!
end R;
```

0.h even though Q is declared in the declarative region of Standard, because R does not mention Q in a with_clause.

8.3 Visibility

Extensions to Ada 83

0.a Declarations with the same defining name as that of a subprogram or entry being defined are nevertheless visible within the subprogram specification or entry declaration.

Wording Changes From Ada 83

0.b The term ‘‘visible by selection’’ is no longer defined. We use the terms ‘‘directly visible’’ and ‘‘visible’’ (among other things). There are only two regions of text that are of interest, here: the region in which a declaration is visible, and the region in which it is directly visible.

0.c Visibility is defined only for declarations.

8.4 Use Clauses

Extensions to Ada 83

0.a The use_type_clause is new to Ada 9X.

Wording Changes From Ada 83

0.b The phrase ‘‘omitting from this set any packages that enclose this place’’ is no longer necessary to avoid making something visible outside its scope, because we explicitly state that the declaration has to be visible in order to be potentially use-visible.

8.5 Renaming Declarations

Wording Changes From Ada 83

0.a The second sentence of RM83-8.5(3), ‘‘At any point where a renaming declaration is visible, the identifier, or operator symbol of this declaration denotes the renamed entity.’’ is incorrect. It doesn’t say directly visible. Also, such an identifier might resolve to something else.

0.b The verbiage about renamings being legal ‘‘only if exactly one...’’, which appears in RM83-8.5(4) (for objects) and RM83-8.5(7) (for subprograms) is removed, because it follows from the normal rules about overload resolution. For language lawyers, these facts are obvious; for programmers, they are irrelevant, since failing these tests is highly unlikely.

8.5.1 Object Renaming Declarations

Wording Changes From Ada 83

The phrase ‘‘subtype ... as defined in a corresponding object declaration, component declaration, or component subtype indication,’’ from RM83-8.5(5), is incorrect in Ada 9X; therefore we removed it. It is incorrect in the case of an object with an indefinite unconstrained nominal subtype. 0.a

8.5.2 Exception Renaming Declarations

8.5.3 Package Renaming Declarations

8.5.4 Subprogram Renaming Declarations

8.5.5 Generic Renaming Declarations

Extensions to Ada 83

Renaming of generic units is new to Ada 9X. It is particularly important for renaming child library units that are generic units. For example, it might be used to rename Numerics.Generic_Elementary_Functions as simply Generic_Elementary_Functions, to match the name for the corresponding Ada-83-based package. 0.a

Wording Changes From Ada 83

The information in RM83-8.6, ‘‘The Package Standard,’’ has been updated for the child unit feature, and moved to Annex A, except for the definition of ‘‘predefined type,’’ which has been moved to 3.2.1. 0.b

8.6 The Context of Overload Resolution

Incompatibilities With Ada 83

The new preference rule for operators of root numeric types is upward incompatible, but only in cases that involved *Beaujolais* effects in Ada 83. Such cases are ambiguous in Ada 9X. 0.a

Extensions to Ada 83

The rule that allows an expected type to match an actual expression of a universal type, in combination with the new preference rule for operators of root numeric types, subsumes the Ada 83 "implicit conversion" rules for universal types. 0.b

Wording Changes From Ada 83

In Ada 83, it is not clear what the ‘‘syntax rules’’ are. AI-00157 states that a certain textual rule is a syntax rule, but it’s still not clear how one tells in general which textual rules are syntax rules. We have solved the problem by stating exactly which rules are syntax rules — the ones that appear under the ‘‘Syntax’’ heading. 0.c

RM83 has a long list of the ‘‘forms’’ of rules that are to be used in overload resolution (in addition to the syntax rules). It is not clear exactly which rules fall under each form. We have solved the problem by explicitly marking all rules that are used in overload resolution. Thus, the list of kinds of rules is unnecessary. It is replaced with some introductory (intentionally vague) text explaining the basic idea of what sorts of rules are overloading rules. 0.d

It is not clear from RM83 what information is embodied in a ‘‘meaning’’ or an ‘‘interpretation.’’ ‘‘Meaning’’ and ‘‘interpretation’’ were intended to be synonymous; we now use the latter only in defining the rules about overload resolution. ‘‘Meaning’’ is used only informally. This clause attempts to clarify what is meant by ‘‘interpretation.’’ 0.e

For example, RM83 does not make it clear that overload resolution is required in order to match subprogram_bodies with their corresponding declarations (and even to tell whether a given subprogram_body is the completion of a previous declaration). Clearly, the information needed to do this is part of the ‘‘interpretation’’ of a subprogram_body. The resolution of such things is defined in terms of the ‘‘expected profile’’ concept. Ada 9X has some new cases where expected profiles are needed — the resolution of P’Access, where P might denote a subprogram, is an example. 0.f

RM83-8.7(2) might seem to imply that an interpretation embodies information about what is denoted by each usage name, but not information about which syntactic category each construct belongs to. However, it seems necessary to include such information, since the Ada grammar is highly ambiguous. For example, X(Y) might be a function_call or an indexed_component, and no context-free/syntactic information can tell the difference. It seems like we should view X(Y) as being, for example, ‘‘interpreted as a function_call’’ (if that’s what overload resolution decides it is). Note that 0.g there are examples where the denotation of each usage name does not imply the syntactic category. However, even if that were not true, it seems that intuitively, the interpretation includes that information. Here's an example:

0.h
```
type T;
type A is access T;
type T is array(Integer range 1..10) of A;
I : Integer := 3;
function F(X : Integer := 7) return A;
Y : A := F(I); -- Ambiguous? (We hope so.)
```

0.i Consider the declaration of Y (a complete context). In the above example, overload resolution can easily determine the declaration, and therefore the entity, denoted by Y, A, F, and I. However, given all of that information, we still don't know whether F(I) is a function_call or an indexed_component whose prefix is a function_call. (In the latter case, it is equivalent to F(7).**all**(I).)

0.j It seems clear that the declaration of Y ought to be considered ambiguous. We describe that by saying that there are two interpretations, one as a function_call, and one as an indexed_component. These interpretations are both acceptable to the overloading rules. Therefore, the complete context is ambiguous, and therefore illegal.

0.k It is the intent that the Ada 9X preference rule for root numeric operators is more locally enforceable than that of RM83-4.6(15). It should also eliminate interpretation shifts due to the addition or removal of a use_clause (the so called *Beaujolais* effect).

0.l RM83-8.7 seems to be missing some complete contexts, such as pragma_argument_associations, declarative_items that are not declarations or representation_clauses, and context_items. We have added these, and also replaced the ‘‘must be determinable’’ wording of RM83-5.4(3) with the notion that the expression of a case_statement is a complete context.

0.m Cases like the Val attribute are now handled using the normal type resolution rules, instead of having special cases that explicitly allow things like ‘‘any integer type.’’

Section 9: Tasks and Synchronization

Wording Changes From Ada 83

The introduction has been rewritten. 0.a

We use the term "concurrent" rather than "parallel" when talking about logically independent execution of threads of control. The term "parallel" is reserved for referring to the situation where multiple physical processors run simultaneously. 0.b

9.1 Task Units and Task Objects

Extensions to Ada 83

The syntax rules for task declarations are modified to allow a known_discriminant_part, and to allow a private part. They are also modified to allow entry_declarations and representation_clauses to be mixed. 0.a

Wording Changes From Ada 83

The syntax rules for tasks have been split up according to task types and single tasks. In particular: The syntax rules for task_declaration and task_specification are removed. The syntax rules for task_type_declaration, single_task_declaration, task_definition and task_item are new. 0.b

The syntax rule for task_body now uses the nonterminal handled_sequence_of_statements. 0.c

The declarative_part of a task_body is now required; that doesn't make any real difference, because a declarative_part can be empty. 0.d

9.2 Task Execution - Task Activation

Wording Changes From Ada 83

We have replaced the term *suspended* with *blocked*, since we didn't want to consider a task blocked when it was simply competing for execution resources. "Suspended" is sometimes used more generally to refer to tasks that are not actually running on some processor, due to the lack of resources. 0.a

This clause has been rewritten in an attempt to improve presentation. 0.b

9.3 Task Dependence - Termination of Tasks

Wording Changes From Ada 83

We have revised the wording to be consistent with the definition of master now given in 7.6.1, ‘‘Completion and Finalization’’. 0.a

Tasks that used to depend on library packages in Ada 83, now depend on the (implicit) task_body of the environment task (see 10.2). Therefore, the environment task has to wait for them before performing library level finalization and terminating the partition. In Ada 83 the requirement to wait for tasks that depended on library packages was not as clear. 0.b

What was "collective termination" is now "collective completion" resulting from selecting terminate_alternatives. This is because finalization still occurs for such tasks, and this happens after selecting the terminate_alternative, but before termination. 0.c

9.4 Protected Units and Protected Objects

Extensions to Ada 83

This entire clause is new; protected units do not exist in Ada 83. 0.a

9.5 Intertask Communication

9.5.1 Protected Subprograms and Protected Actions

9.5.2 Entries and Accept Statements

Extensions to Ada 83

0.a The syntax rule for entry_body is new.

0.b Accept_statements can now have exception_handlers.

9.5.3 Entry Calls

9.5.4 Requeue Statements

Extensions to Ada 83

0.a The requeue_statement is new.

9.6 Delay Statements, Duration, and Time

Inconsistencies With Ada 83

0.a For programs that raise Time_Error on "+" or "–" in Ada 83,the exception might be deferred until a call on Split or Year_Number, or might not be raised at all (if the offending time is never Split after being calculated). This should not affect typical programs, since they deal only with times corresponding to the relatively recent past or near future.

Extensions to Ada 83

0.b The syntax rule for delay_statement is modified to allow delay_until_statements.

0.c The type Time may represent dates with year numbers outside of Year_Number. Therefore, the operations "+" and "–" need only raise Time_Error if the result is not representable in Time (or Duration); also, Split or Year will now raise Time_Error if the year number is outside of Year_Number. This change is intended to simplify the implementation of "+" and "–" (allowing them to depend on overflow for detecting when to raise Time_Error) and to allow local timezone information to be considered at the time of Split rather than Clock (depending on the implementation approach). For example, in a POSIX environment, it is natural for the type Time to be based on GMT, and the results of procedure Split (and the functions Year, Month, Day, and Seconds) to depend on local time zone information. In other environments, it is more natural for the type Time to be based on the local time zone, with the results of Year, Month, Day, and Seconds being pure functions of their input.

0.d We anticipate that implementations will provide child packages of Calendar to provide more explicit control over time zones and other environment-dependent time-related issues. These would be appropriate for standardization in a given environment (such as POSIX).

9.7 Select Statements

Extensions to Ada 83

0.a Asynchronous_select is new.

9.7.1 Selective Accept

Wording Changes From Ada 83

0.a The name of selective_wait was changed to selective_accept to better describe what is being waited for. We kept select_alternative as is, because selective_accept_alternative was too easily confused with accept_alternative.

9.7.2 Timed Entry Calls

Wording Changes From Ada 83

This clause comes before the one for Conditional Entry Calls, so we can define conditional entry calls in terms of timed entry calls. 0.a

9.7.3 Conditional Entry Calls

Wording Changes From Ada 83

This clause comes after the one for Timed Entry Calls, so we can define conditional entry calls in terms of timed entry calls. We do that so that an "expiration time" is defined for both, thereby simplifying the definition of what happens on a requeue-with-abort. 0.a

9.7.4 Asynchronous Transfer of Control

Extensions to Ada 83

Asynchronous_select is new. 0.a

9.8 Abort of a Task - Abort of a Sequence of Statements

Wording Changes From Ada 83

This clause has been rewritten to accommodate the concept of aborting the execution of a construct, rather than just of a task. 0.a

9.9 Task and Entry Attributes

9.10 Shared Variables

9.11 Example of Tasking and Synchronization

Chg839X;5.95

Section 10: Program Structure and Compilation Issues

Wording Changes From Ada 83

The section organization mentioned above is different from that of RM83. 0.a

10.1 Separate Compilation

Wording Changes From Ada 83

The interactions between language issues and environmental issues are left open in Ada 9X. The environment concept is new. In Ada 83, the concept of the program library, for example, appeared to be quite concrete, although the rules had no force, since implementations could get around them simply by defining various mappings from the concept of an Ada program library to whatever data structures were actually stored in support of separate compilation. Indeed, implementations were encouraged to do so. 0.a

In RM83, it was unclear which was the official definition of ‘‘program unit.’’ Definitions appeared in RM83-5, 6, 7, and 9, but not 12. Placing it here seems logical, since a program unit is sort of a potential compilation unit. 0.b

10.1.1 Compilation Units - Library Units

Extensions to Ada 83

The syntax rule for library_item is modified to allow the reserved word **private** before a library_unit_declaration. 0.a

Children (other than children of Standard) are new in Ada 9X. 0.b

Library unit renaming is new in Ada 9X. 0.c

Wording Changes From Ada 83

Standard is considered a library unit in Ada 9X. This simplifies the descriptions, since it implies that the parent of each library unit is a library unit. (Standard itself has no parent, of course.) As in Ada 83, the language does not define any way to recompile Standard, since the name given in the declaration of a library unit is always interpreted in relation to Standard. That is, an attempt to compile a package Standard would result in Standard.Standard. 0.d

10.1.2 Context Clauses - With Clauses

Extensions to Ada 83

The syntax rule for with_clause is modified to allow expanded name notation. 0.a

A use_clause in a context_clause may be for a package (or type) nested in a library package. 0.b

Wording Changes From Ada 83

The syntax rule for context_clause is modified to more closely reflect the semantics. The Ada 83 syntax rule implies that the use_clauses that appear immediately after a particular with_clause are somehow attached to that with_clause, which is not true. The new syntax allows a use_clause to appear first, but that is prevented by a textual rule that already exists in Ada 83. 0.c

The concept of ‘‘scope of a with_clause’’ (which is a region of text) replaces RM83’s notion of ‘‘apply to’’ (a with_clause applies to a library_item) The visibility rules are interested in a region of text, not in a set of compilation units. 0.d

No need to define ‘‘apply to’’ for use_clauses. Their semantics are fully covered by the ‘‘scope (of a use_clause)’’ definition in 8.4. 0.e

10.1.3 Subunits of Compilation Units

Extensions to Ada 83

Subunits of the same ancestor library unit are no longer restricted to have distinct identifiers. Instead, we require only that the full expanded names be distinct. 0.a

10.1.4 The Compilation Process

10.1.5 Pragmas and Program Units

10.1.6 Environment-Level Visibility Rules

Wording Changes From Ada 83

0.a The special visibility rules that apply within a parent_unit_name or a context_clause, and within a pragma that appears at the place of a compilation_unit are clarified.

0.b Note that a context_clause is not part of any declarative region.

0.c We considered making the visibility rules within parent_unit_names and context_clauses follow from the context of compilation. However, this attempt failed for various reasons. For example, it would require use_clauses in context_clauses to be within the declarative region of Standard, which sounds suspiciously like a kludge. And we would still need a special rule to prevent seeing things (in our own context_clause) that were with-ed by our parent, etc.

10.2 Program Execution

Extensions to Ada 83

The concept of partitions is new to Ada 9X. 0.a

A main subprogram is now optional. The language-defined restrictions on main subprograms are relaxed. 0.b

Wording Changes From Ada 83

Ada 9X uses the term ‘‘main subprogram’’ instead of Ada 83’s ‘‘main program’’ (which was inherited from Pascal). This is done to avoid confusion — a main subprogram is a subprogram, not a program. The program as a whole is an entirely different thing. 0.c

10.2.1 Elaboration Control

Extensions to Ada 83

The concepts of preelaborability and purity are new to Ada 9X. The Elaborate_All, Elaborate_Body, Preelaborate, and Pure pragmas are new to Ada 9X. 0.a

Pragmas Elaborate are allowed to be mixed in with the other things in the context_clause — in Ada 83, they were required to appear last. 0.b

Section 11: Exceptions

Wording Changes From Ada 83

We are more explicit about the difference between an exception and an occurrence of an exception. This is necessary because we now have a type (Exception_Occurrence) that represents exception occurrences, so the program can manipulate them. Furthermore, we say that when an exception is propagated, it is the same occurrence that is being propagated (as opposed to a new occurrence of the same exception). The same issue applies to a re-raise statement. In order to understand these semantics, we have to make this distinction. 0.a

11.1 Exception Declarations

Inconsistencies With Ada 83

The exception Numeric_Error is now defined in the Obsolescent features Annex, as a rename of Constraint_Error. All checks that raise Numeric_Error in Ada 83 instead raise Constraint_Error in Ada 9X. To increase upward compatibility, we also changed the rules to allow the same exception to be named more than once by a given handler. Thus, ‘‘**when** Constraint_Error | Numeric_Error =>’’ will remain legal in Ada 9X, even though Constraint_Error and Numeric_Error now denote the same exception. However, it will not be legal to have separate handlers for Constraint_Error and Numeric_Error. This change is inconsistent in the rare case that an existing program explicitly raises Numeric_Error at a point where there is a handler for Constraint_Error; the exception will now be caught by that handler. 0.a

Wording Changes From Ada 83

We explicitly define elaboration for exception_declarations. 0.b

11.2 Exception Handlers

Extensions to Ada 83

The syntax rule for exception_handler is modified to allow a choice_parameter_specification. 0.a

Different choices of the same exception_handler may cover the same exception. This allows for ‘‘when Numeric_Error | Constraint_Error =>’’ even though Numeric_Error is a rename of Constraint_Error. This also allows one to ‘‘with’’ two different I/O packages, and then write, for example, ‘‘when Ada.Text_IO.Data_Error | My_Seq_IO.Data_Error =>’’ even though these might both be renames of the same exception. 0.b

Wording Changes From Ada 83

The syntax rule for handled_sequence_of_statements is new. These are now used in all the places where handlers are allowed. This obviates the need to explain (in Sections 5, 6, 7, and 9) what portions of the program are handled by the handlers. Note that there are more such cases in Ada 9X. 0.c

The syntax rule for choice_parameter_specification is new. 0.d

11.3 Raise Statements

Wording Changes From Ada 83

The fact that the name in a raise_statement has to denote an exception is not clear from RM83. Clearly that was the intent, since the italicized part of the syntax rules so indicate, but there was no explicit rule. RM83-1.5(11) doesn't seem to give the italicized parts of the syntax any force. 0.a

11.4 Exception Handling

11.4.1 The Package Exceptions

Extensions to Ada 83

The Identity attribute of exceptions is new, as is the package Exceptions. 0.a

11.4.2 Example of Exception Handling

Wording Changes From Ada 83

0.a The sections labeled ‘‘Exceptions Raised During ...’’ are subsumed by this clause, and by parts of Section 9.

11.5 Suppressing Checks

Extensions to Ada 83

0.a A pragma Suppress is allowed as a configuration pragma. A pragma Suppress without a name is allowed in a package_specification.

0.b Additional check names are added. We allow implementations to define their own checks.

Wording Changes From Ada 83

0.c We define the checks in a distributed manner. Therefore, the long list of what checks apply to what is merely a NOTE.

0.d We have removed the detailed rules about what is allowed in a pragma Suppress, and allow implementations to invent their own. The RM83 rules weren’t quite right, and such a change is necessary anyway in the presence of implementation-defined checks.

0.e We make it clear that the difference between a Range_Check and an Overflow_Check is fuzzy. This was true in Ada 83, given RM83-11.6, but it was not clear. We considered removing Overflow_Check from the language or making it obsolescent, just as we did for Numeric_Error. However, we kept it for upward compatibility, and because it may be useful on machines where range checking costs more than overflow checking, but overflow checking still costs something. Different compilers will suppress different checks when asked to suppress Overflow_Check — the non-uniformity in this case is not harmful, and removing it would have a serious impact on optimizers.

0.f Under Access_Check, dereferences cover the cases of selected_component, indexed_component, slice, and attribute that are listed in RM83, as well as the new explicit_dereference, which was included in selected_component in RM83.

11.6 Exceptions and Optimization

Wording Changes From Ada 83

0.a RM83-11.6 was unclear. It has been completely rewritten here; we hope this version is clearer. Here’s what happened to each paragraph of RM83-11.6:

0.b • Paragraphs 1 and 2 contain no semantics; they are merely pointing out that anything goes if the canonical semantics is preserved. We have similar introductory paragraphs, but we have tried to clarify that these are not granting any ‘‘extra’’ permission beyond what the rest of the document allows.

0.c • Paragraphs 3 and 4 are reflected in the ‘‘extra permission to reorder actions’’. Note that this permission now allows the reordering of assignments in many cases.

0.d • Paragraph 5 is moved to 4.5, ‘‘Operators and Expression Evaluation’’, where operator association is discussed. Hence, this is no longer an ‘‘extra permission’’ but is part of the canonical semantics.

0.e • Paragraph 6 now follows from the general permission to store out-of-range values for unconstrained subtypes. Note that the parameters and results of all the predefined operators of a type are of the unconstrained subtype of the type.

0.f • Paragraph 7 is reflected in the ‘‘extra permission to avoid raising exceptions’’.

0.g We moved clause 11.5, ‘‘Suppressing Checks’’ from after 11.6 to before 11.6, in order to preserve the famous number ‘‘11.6’’ (given the changes to earlier clauses in Section 11).

Section 12: Generic Units

12.1 Generic Declarations

Extensions to Ada 83

The syntax rule for generic_formal_parameter_declaration is modified to allow the reserved words **tagged** and **abstract**, to allow formal derived types, and to allow formal packages. 0.a

Use_clauses are allowed in generic_formal_parts. This is necessary in order to allow a use_clause within a formal part to provide direct visibility of declarations within a generic formal package. 0.b

Wording Changes From Ada 83

The syntax for generic_formal_parameter_declaration and formal_type_definition is split up into more named categories. The rules for these categories are moved to the appropriate clauses and subclauses. The names of the categories are changed to be more intuitive and uniform. For example, we changed generic_parameter_declaration to generic_formal_parameter_declaration, because the thing it declares is a generic formal, not a generic. In the others, we abbreviate ‘‘generic_formal’’ to just ‘‘formal’’. We can’t do that for generic_formal_parameter_declaration, because of confusion with normal formal parameters of subprograms. 0.c

12.2 Generic Bodies

12.3 Generic Instantiation

Inconsistencies With Ada 83

In Ada 83, all explicit actuals are evaluated before all defaults, and the defaults are evaluated in the order of the formal declarations. This ordering requirement is relaxed in Ada 9X. 0.a

Incompatibilities With Ada 83

We have attempted to remove every violation of the contract model. Any remaining contract model violations should be considered bugs in the RM9X. The unfortunate property of reverting to the predefined operators of the actual types is retained for upward compatibility. (Note that fixing this would require subtype conformance rules.) However, tagged types do not revert in this sense. 0.b

Extensions to Ada 83

The syntax rule for explicit_generic_actual_parameter is modified to allow a *package_instance_*name. 0.c

Wording Changes From Ada 83

The fact that named associations cannot be used for two formal subprograms with the same defining name is moved to AARM-only material, because it is a ramification of other rules, and because it is not of interest to the average user. 0.d

The rule that ‘‘An explicit explicit_generic_actual_parameter shall not be supplied more than once for a given generic_formal_parameter’’ seems to be missing from RM83, although it was clearly the intent. 0.e

In the explanation that the instance is a copy of the template, we have left out RM83-12.3(5)’s ‘‘apart from the generic formal part’’, because it seems that things in the formal part still need to exist in instances. This is particularly true for generic formal packages, where you’re sometimes allowed to reach in and denote the formals of the formal package from outside it. This simplifies the explanation of what each name in an instance denotes: there are just two cases: the declaration can be inside or outside (where inside needs to include the generic unit itself). Note that the RM83 approach of listing many cases (see RM83-12.5(5-14)) would have become even more unwieldy with the addition of generic formal packages, and the declarations that occur therein. 0.f

We have corrected the definition of the elaboration of a generic_instantiation (RM83-12.3(17)); we don’t elaborate entities, and the instance is not ‘‘implicit.’’ 0.g

In RM83, there is a rule saying the formal and actual shall match, and then there is much text defining what it means to match. Here, we simply state all the latter text as rules. For example, ‘‘A formal foo is matched by an actual greenish bar’’ becomes ‘‘For a formal foo, the actual shall be a greenish bar.’’ This is necessary to split the Name Resolution Rules from the Legality Rules. Besides, there’s really no need to define the concept of matching for generic parameters. 0.h

12.4 Formal Objects

Extensions to Ada 83

0.a In Ada 83, it is forbidden to pass a (nongeneric) formal parameter of mode **out**, or a subcomponent thereof, to a generic formal object of mode **in out**. This restriction is removed in Ada 9X.

Wording Changes From Ada 83

0.b We make ‘‘mode’’ explicit in the syntax. RM83 refers to the mode without saying what it is. This is also more uniform with the way (nongeneric) formal parameters are defined.

0.c We considered allowing mode **out** in Ada 9X, for uniformity with (nongeneric) formal parameters. The semantics would be identical for modes **in out** and **out**. (Note that generic formal objects of mode **in out** are passed by reference. Note that for (nongeneric) formal parameters that are allowed to be passed by reference, the semantics of **in out** and **out** is the same. The difference might serve as documentation. The same would be true for generic formal objects, if **out** were allowed, so it would be consistent.) We decided not to make this change, because it does not produce any important benefit, and any change has some cost.

12.5 Formal Types

Wording Changes From Ada 83

0.a RM83 has separate sections ‘‘Generic Formal Xs’’ and ‘‘Matching Rules for Formal Xs’’ (for various X’s) with most of the text redundant between the two. We have combined the two in order to reduce the redundancy. In RM83, there is no ‘‘Matching Rules for Formal Types’’ section; nor is there a ‘‘Generic Formal Y Types’’ section (for Y = Private, Scalar, Array, and Access). This causes, for example, the duplication across all the ‘‘Matching Rules for Y Types’’ sections of the rule that the actual passed to a formal type shall be a subtype; the new organization avoids that problem.

0.b The matching rules are stated more concisely.

0.c We no longer consider the multiplying operators that deliver a result of type *universal_fixed* to be predefined for the various types; there is only one of each in package Standard. Therefore, we need not mention them here as RM83 had to.

12.5.1 Formal Private and Derived Types

Incompatibilities With Ada 83

0.a Ada 83 does not have unknown_discriminant_parts, so it allows indefinite subtypes to be passed to definite formals, and applies a legality rule to the instance body. This is a contract model violation. Ada 9X disallows such cases at the point of the instantiation. The workaround is to add (<>) as the discriminant_part of any formal subtype if it is intended to be used with indefinite actuals. If that’s the intent, then there can’t be anything in the generic body that would require a definite subtype.

0.b The check for discriminant subtype matching is changed from a run-time check to a compile-time check.

12.5.2 Formal Scalar Types

12.5.3 Formal Array Types

Incompatibilities With Ada 83

0.a The check for matching of component subtypes and index subtypes or index ranges is changed from a run-time check to a compile-time check. The Ada 83 rule that ‘‘If the component type is not a scalar type, then the component subtypes shall be either both constrained or both unconstrained’’ is removed, since it is subsumed by static matching. Likewise, the rules requiring that component types be the same is subsumed.

12.5.4 Formal Access Types

Incompatibilities With Ada 83

0.a The check for matching of designated subtypes is changed from a run-time check to a compile-time check. The Ada 83 rule that ‘‘If the designated type is other than a scalar type, then the designated subtypes shall be either both constrained or both unconstrained’’ is removed, since it is subsumed by static matching.

Extensions to Ada 83

Formal access-to-subprogram subtypes and formal general access types are new concepts. 0.b

12.6 Formal Subprograms

12.7 Formal Packages

Extensions to Ada 83

Formal packages are new to Ada 9X. 0.a

12.8 Example of a Generic Package

Chg839X;5.95

Section 13: Representation Issues

Wording Changes From Ada 83

The clauses of this section have been reorganized. This was necessary to preserve a logical order, given the new Ada 9X semantics given in this section. 0.a

13.1 Representation Items

Incompatibilities With Ada 83

It is now illegal for a representation item to cause a derived by-reference type to have a different record layout from its parent. This is necessary for by-reference parameter passing to be feasible. This only affects programs that specify the representation of types derived from types containing tasks; most by-reference types are new to Ada 9X. For example, if A1 is an array of tasks, and A2 is derived from A1, it is illegal to apply a pragma Pack to A2. 0.a

Extensions to Ada 83

Ada 9X allows additional representation_clauses for objects. 0.b

Wording Changes From Ada 83

The syntax rule for type_representation_clause is removed; the right-hand side of that rule is moved up to where it was used, in representation_clause. There are two references to ‘‘type representation clause’’ in RM83, both in Section 13; these have been reworded. 0.c

We have defined a new term ‘‘representation item,’’ which includes both representation_clauses and representation pragmas, as well as component_clauses. This is convenient because the rules are almost identical for all three. 0.d

All of the forcing occurrence stuff has been moved into its own subclause (see 13.14), and rewritten to use the term ‘‘freezing’’. 0.e

RM83-13.1(10) requires implementation-defined restrictions on representation items to be enforced at compile time. However, that is impossible in some cases. If the user specifies a junk (nonstatic) address in an address clause, and the implementation chooses to detect the error (for example, using hardware memory management with protected pages), then it’s clearly going to be a run-time error. It seems silly to call that ‘‘semantics’’ rather than ‘‘a restriction.’’ 0.f

RM83-13.1(10) tries to pretend that representation_clauses don’t affect the semantics of the program. One counter-example is the Small clause. Ada 9X has more counter-examples. We have noted the opposite above. 0.g

Some of the more stringent requirements are moved to C.2, ‘‘Required Representation Support’’. 0.h

13.2 Pragma Pack

13.3 Representation Attributes

Wording Changes From Ada 83

The intended meaning of the various attributes, and their attribute_definition_clauses, is more explicit. 0.a

The address_clause has been renamed to at_clause and moved to Annex J, ‘‘Obsolescent Features’’. One can use an Address clause (‘‘for T’Address **use** ...;’’) instead. 0.b

The attributes defined in RM83-13.7.3 are moved to Annex G, A.5.3, and A.5.4. 0.c

Wording Changes From Ada 83

The nonnegative part is missing from RM83 (for mod_clauses, nee alignment_clauses, which are an obsolete version of Alignment clauses). 0.d

Wording Changes From Ada 83

The requirement for a nonnegative value in a Size clause was not in RM83, but it’s hard to see how it would make sense. For uniformity, we forbid negative sizes, rather than letting implementations define their meaning. 0.e

Extensions to Ada 83

The syntax rule for length_clause is replaced with the new syntax rule for attribute_definition_clause, and it is modified to allow a name (as well as an expression). 0.f

Wording Changes From Ada 83

0.g The syntax rule for attribute_definition_clause now requires that the prefix of the attribute be a local_name; in Ada 83 this rule was stated in the text.

0.h In Ada 83, the relationship between a representation_clause specifying a certain aspect and an attribute that queried that aspect was unclear. In Ada 9X, they are the same, except for certain explicit exceptions.

13.4 Enumeration Representation Clauses

Extensions to Ada 83

0.a As in other similar contexts, Ada 9X allows expressions of any integer type, not just expressions of type *universal_ integer*, for the component expressions in the enumeration_aggregate. The preference rules for the predefined operators of *root_integer* eliminate any ambiguity.

0.b For portability, we now require that the default coding for an enumeration type be the ‘‘obvious’’ coding using position numbers. This is satisfied by all known implementations.

13.5 Record Layout

13.5.1 Record Representation Clauses

Wording Changes From Ada 83

0.a The alignment_clause has been renamed to mod_clause and moved to Annex J, ‘‘Obsolescent Features’’.

0.b We have clarified that implementation-defined component names have to be in the form of an attribute_reference of a component or of the first subtype itself; surely Ada 83 did not intend to allow arbitrary identifiers.

0.c The RM83-13.4(7) wording incorrectly allows components in non-variant records to overlap. We have corrected that oversight.

13.5.2 Storage Place Attributes

13.5.3 Bit Ordering

Extensions to Ada 83

0.a The Bit_Order attribute is new to Ada 9X.

13.6 Change of Representation

13.7 The Package System

Wording Changes From Ada 83

0.a Much of the content of System is standardized, to provide more uniformity across implementations. Implementations can still add their own declarations to System, but are encouraged to do so via children of System.

0.b Some of the named numbers are defined more explicitly in terms of the standard numeric types.

0.c The pragmas System_Name, Storage_Unit, and Memory_Size are no longer defined by the language. However, the corresponding declarations in package System still exist. Existing implementations may continue to support the three pragmas as implementation-defined pragmas, if they so desire.

0.d Priority semantics, including subtype Priority, have been moved to the Real Time Annex.

13.7.1 The Package System.Storage_Elements

13.7.2 The Package System.Address_To_Access_Conversions

13.8 Machine Code Insertions

Extensions to Ada 83

Machine code functions are allowed in Ada 9X; in Ada 83, only procedures were allowed. 0.a

Wording Changes From Ada 83

The syntax for code_statement is changed to say ‘‘qualified_expression’’ instead of ‘‘subtype_mark’record_ aggregate’’. Requiring the type of each instruction to be a record type is overspecification. 0.b

13.9 Unchecked Type Conversions

13.9.1 Data Validity

Wording Changes From Ada 83

In order to reduce the amount of erroneousness, we separate the concept of an undefined value into objects with invalid representation (scalars only) and abnormal objects. 0.a

Reading an object with an invalid representation is a bounded error rather than erroneous; reading an abnormal object is still erroneous. In fact, the only safe thing to do to an abnormal object is to assign to the object as a whole. 0.b

13.9.2 The Valid Attribute

Extensions to Ada 83

X’Valid is new in Ada 9X. 0.a

13.10 Unchecked Access Value Creation

13.11 Storage Management

Extensions to Ada 83

User-defined storage pools are new to Ada 9X. 0.a

Wording Changes From Ada 83

Ada 83 had a concept called a ‘‘collection,’’ which is similar to what we call a storage pool. All access types in the same derivation class shared the same collection. In Ada 9X, all access types in the same derivation class share the same storage pool, but other (unrelated) access types can also share the same storage pool, either by default, or as specified by the user. A collection was an amorphous collection of objects; a storage pool is a more concrete concept — hence the different name. 0.b

RM83 states the erroneousness of reading or updating deallocated objects incorrectly by missing various cases. 0.c

13.11.1 The Max_Size_In_Storage_Elements Attribute

13.11.2 Unchecked Storage Deallocation

13.11.3 Pragma Controlled

Wording Changes From Ada 83

0.a Ada 83 used the term ‘‘automatic storage reclamation’’ to refer to what is known traditionally as ‘‘garbage collection’’. Because of the existence of storage pools (see 13.11), we need to distinguish this from the storage reclamation that might happen upon leaving a master. Therefore, we now use the term ‘‘garbage collection’’ in its normal computer-science sense. This has the additional advantage of making our terminology more accessible to people outside the Ada world.

13.12 Pragma Restrictions

Extensions to Ada 83

0.a Pragma Restrictions is new to Ada 9X.

13.13 Streams

Extensions to Ada 83

0.a Streams are new in Ada 9X.

13.13.1 The Package Streams

13.13.2 Stream-Oriented Attributes

13.14 Freezing Rules

Incompatibilities With Ada 83

0.a RM83 defines a forcing occurrence of a type as follows: ‘‘A forcing occurrence is any occurrence [of the name of the type, subtypes of the type, or types or subtypes with subcomponents of the type] other than in a type or subtype declaration, a subprogram specification, an entry declaration, a deferred constant declaration, a pragma, or a representation_clause for the type itself. In any case, an occurrence within an expression is always forcing.’’

0.b It seems like the wording allows things like this:

0.c
```
type A is array(Integer range 1..10) of Boolean;
subtype S is Integer range A'Range;
    -- not forcing for A
```

0.d Occurrences within pragmas can cause freezing in Ada 9X. (Since such pragmas are ignored in Ada 83, this will probably fix more bugs than it causes.)

Extensions to Ada 83

0.e In Ada 9X, generic_formal_parameter_declarations do not normally freeze the entities from which they are defined. For example:

0.f
```
package Outer is
    type T is tagged limited private;
    generic
        type T2 is
            new T with private; -- Does not freeze T
                                -- in Ada 9X.
    package Inner is
        ...
    end Inner;
private
    type T is ...;
end Outer;
```

This is important for the usability of generics. The above example uses the Ada 9X feature of formal derived types. Examples using the kinds of formal parameters already allowed in Ada 83 are well known. See, for example, comments 83-00627 and 83-00688. The extensive use expected for formal derived types makes this issue even more compelling than described by those comments. Unfortunately, we are unable to solve the problem that explicit_generic_actual_parameters cause freezing, even though a package equivalent to the instance would not cause freezing. This is primarily because such an equivalent package would have its body in the body of the containing program unit, whereas an instance has its body right there. 0.g

Wording Changes From Ada 83

The concept of freezing is based on Ada 83's concept of ''forcing occurrences.'' The first freezing point of an entity corresponds roughly to the place of the first forcing occurrence, in Ada 83 terms. The reason for changing the terminology is that the new rules do not refer to any particular ''occurrence'' of a name of an entity. Instead, we refer to ''uses'' of an entity, which are sometimes implicit. 0.h

In Ada 83, forcing occurrences were used only in rules about representation_clauses. We have expanded the concept to cover private types, because the rules stated in RM83-7.4.1(4) are almost identical to the forcing occurrence rules. 0.i

The Ada 83 rules are changed in Ada 9X for the following reasons: 0.j

- The Ada 83 rules do not work right for subtype-specific aspects. In an earlier version of Ada 9X, we considered allowing representation items to apply to subtypes other than the first subtype. This was part of the reason for changing the Ada 83 rules. However, now that we have dropped that functionality, we still need the rules to be different from the Ada 83 rules. 0.k
- The Ada 83 rules do not achieve the intended effect. In Ada 83, either with or without the AIs, it is possible to force the compiler to generate code that references uninitialized dope, or force it to detect erroneousness and exception raising at compile time. 0.l
- It was a goal of Ada 83 to avoid uninitialized access values. However, in the case of deferred constants, this goal was not achieved. 0.m
- The Ada 83 rules are not only too weak — they are also too strong. They allow loopholes (as described above), but they also prevent certain kinds of default_expressions that are harmless, and certain kinds of generic_declarations that are both harmless and very useful. 0.n
- Ada 83 had a case where a representation_clause had a strong effect on the semantics of the program — 'Small. This caused certain semantic anomalies. There are more cases in Ada 9X, because the attribute_representation_clause has been generalized. 0.o

The Standard Libraries

Chg839X;5.95

Annex A
(normative)

Predefined Language Environment

Wording Changes From Ada 83

Many of Ada 83's language-defined library units are now children of Ada or System. For upward compatibility, these are renamed as root library units (see J.1). 0.a

The order and lettering of the annexes has been changed. 0.b

A.1 The Package Standard

Extensions to Ada 83

Package Standard is declared to be pure. 0.a

Wording Changes From Ada 83

Numeric_Error is made obsolescent. 0.b

The declarations of Natural and Positive are moved to just after the declaration of Integer, so that "**" can refer to Natural without a forward reference. There's no real need to move Positive, too — it just came along for the ride. 0.c

A.2 The Package Ada

Extensions to Ada 83

This clause is new to Ada 9X. 0.a

A.3 Character Handling

Extensions to Ada 83

This clause is new to Ada 9X. 0.a

A.3.1 The Package Characters

A.3.2 The Package Characters.Handling

A.3.3 The Package Characters.Latin_1

A.4 String Handling

Extensions to Ada 83

This clause is new to Ada 9X. 0.a

A.4.1 The Package Strings

A.4.2 The Package Strings.Maps

A.4.3 Fixed-Length String Handling

A.4.4 Bounded-Length String Handling

A.4.5 Unbounded-Length String Handling

A.4.6 String-Handling Sets and Mappings

A.4.7 Wide_String Handling

A.5 The Numerics Packages

Extensions to Ada 83

0.a Numerics and its children were not predefined in Ada 83.

A.5.1 Elementary Functions

Wording Changes From Ada 83

0.a The semantics of Numerics.Generic_Elementary_Functions differs from Generic_Elementary_Functions as defined in ISO/IEC DIS 11430 (for Ada 83) in the following ways:

0.b • The generic package is a child unit of the package defining the Argument_Error exception.

0.c • DIS 11430 specified names for the nongeneric equivalents, if provided. Here, those nongeneric equivalents are required.

0.d • Implementations are not allowed to impose an optional restriction that the generic actual parameter associated with Float_Type be unconstrained. (In view of the ability to declare variables of subtype Float_Type'Base in implementations of Numerics.Generic_Elementary_Functions, this flexibility is no longer needed.)

0.e • The sign of a prescribed zero result at the origin of the odd functions is specified, when Float_Type'Signed_Zeros is True. This conforms with recommendations of Kahan and other numerical analysts.

0.f • The dependence of Arctan and Arccot on the sign of a parameter value of zero is tied to the value of Float_Type'Signed_Zeros.

0.g • Sqrt is prescribed to yield a result of one when its parameter has the value one. This guarantee makes it easier to achieve certain prescribed results of the complex elementary functions (see G.1.2, ‘‘Complex Elementary Functions’’).

0.h • Conformance to accuracy requirements is conditional.

A.5.2 Random Number Generation

A.5.3 Attributes of Floating Point Types

Incompatibilities With Ada 83

The Epsilon and Mantissa attributes of floating point types are removed from the language and replaced by Model_Epsilon and Model_Mantissa, which may have different values (as a result of changes in the definition of model numbers); the replacement of one set of attributes by another is intended to convert what would be an inconsistent change into an incompatible change. 0.a

The Emax, Small, Large, Safe_Emax, Safe_Small, and Safe_Large attributes of floating point types are removed from the language. Small and Safe_Small are collectively replaced by Model_Small, which is functionally equivalent to Safe_Small, though it may have a slightly different value. The others are collectively replaced by Safe_First and Safe_Last. Safe_Last is functionally equivalent to Safe_Large, though it may have a different value; Safe_First is comparable to the negation of Safe_Large but may differ slightly from it as well as from the negation of Safe_Last. Emax and Safe_Emax had relatively few uses in Ada 83; T'Safe_Emax can be computed in the revised language as Integer'Min(T'Exponent(T'Safe_First), T'Exponent(T'Safe_Last)). 0.b

Implementations are encouraged to eliminate the incompatibilities discussed here by retaining the old attributes, during a transition period, in the form of implementation-defined attributes with their former values. 0.c

Extensions to Ada 83

The Model_Emin attribute is new. It is conceptually similar to the negation of Safe_Emax attribute of Ada 83, adjusted for the fact that the model numbers now have the hardware radix. It is a fundamental determinant, along with Model_Mantissa, of the set of model numbers of a type (see G.2.1). 0.d

The Denorm and Signed_Zeros attributes are new, as are all of the primitive function attributes. 0.e

A.5.4 Attributes of Fixed Point Types

Incompatibilities With Ada 83

The Mantissa, Large, Safe_Small, and Safe_Large attributes of fixed point types are removed from the language. 0.a

Implementations are encouraged to eliminate the resulting incompatibility by retaining these attributes, during a transition period, in the form of implementation-defined attributes with their former values. 0.b

Extensions to Ada 83

The Machine_Radix attribute is now allowed for fixed point types. It is also specifiable in an attribute definition clause (see F.1). 0.c

A.6 Input-Output

Inconsistencies With Ada 83

The introduction of Append_File as a new element of the enumeration type File_Mode in Sequential_IO and Text_IO, and the introduction of several new declarations in Text_IO, may result in name clashes in the presence of **use** clauses. 0.a

Extensions to Ada 83

Text_IO enhancements (Get_Immediate, Look_Ahead, Standard_Error, Modular_IO, Decimal_IO), Wide_Text_IO, and the stream input-output facilities are new in Ada 9X. 0.b

Wording Changes From Ada 83

RM83-14.6, "Low Level Input-Output," is removed. This has no semantic effect, since the package was entirely implementation defined, nobody actually implemented it, and if they did, they can always provide it as a vendor-supplied package. 0.c

A.7 External Files and File Objects

A.8 Sequential and Direct Files

A.8.1 The Generic Package Sequential_IO

Incompatibilities With Ada 83

0.a The new enumeration element Append_File may introduce upward incompatibilities. It is possible that a program based on the assumption that File_Mode'Last = Out_File will be illegal (e.g., case statement choice coverage) or execute with a different effect in Ada 9X.

A.8.2 File Management

A.8.3 Sequential Input-Output Operations

A.8.4 The Generic Package Direct_IO

A.8.5 Direct Input-Output Operations

A.9 The Generic Package Storage_IO

A.10 Text Input-Output

Extensions to Ada 83

0.a Append_File is new in Ada 9X.

A.10.1 The Package Text_IO

Incompatibilities With Ada 83

0.a Append_File is a new element of enumeration type File_Mode.

Extensions to Ada 83

0.b Get_Immediate, Look_Ahead, the subprograms for dealing with standard error, the type File_Access and its associated subprograms, and the generic packages Modular_IO and Decimal_IO are new in Ada 9X.

A.10.2 Text File Management

A.10.3 Default Input, Output, and Error Files

A.10.4 Specification of Line and Page Lengths

A.10.5 Operations on Columns, Lines, and Pages

A.10.6 Get and Put Procedures

A.10.7 Input-Output of Characters and Strings

A.10.8 Input-Output for Integer Types

A.10.9 Input-Output for Real Types

A.10.10 Input-Output for Enumeration Types

A.11 Wide Text Input-Output

Extensions to Ada 83

Support for Wide_Character and Wide_String I/O is new in Ada 9X. 0.a

A.12 Stream Input-Output

A.12.1 The Package Streams.Stream_IO

A.12.2 The Package Text_IO.Text_Streams

A.12.3 The Package Wide_Text_IO.Text_Streams

A.13 Exceptions in Input-Output

A.14 File Sharing

A.15 The Package Command_Line

Extensions to Ada 83

This clause is new in Ada 9X. 0.a

Annex B
(normative)

Interface to Other Languages

Extensions to Ada 83

Much of the functionality in this Annex is new to Ada 9X. 0.a

Wording Changes From Ada 83

This Annex contains what used to be RM83-13.8. 0.b

B.1 Interfacing Pragmas

Extensions to Ada 83

Interfacing pragmas are new to Ada 9X. Pragma Import replaces Ada 83's pragma Interface. Existing implementations can continue to support pragma Interface for upward compatibility. 0.a

B.2 The Package Interfaces

B.3 Interfacing with C

B.3.1 The Package Interfaces.C.Strings

B.3.2 The Generic Package Interfaces.C.Pointers

B.4 Interfacing with COBOL

B.5 Interfacing with Fortran

Annex C
(normative)

Systems Programming

Extensions to Ada 83

This Annex is new to Ada 9X. 0.a

C.1 Access to Machine Operations

C.2 Required Representation Support

C.3 Interrupt Support

C.3.1 Protected Procedure Handlers

C.3.2 The Package Interrupts

C.4 Preelaboration Requirements

C.5 Pragma Discard_Names

C.6 Shared Variable Control

Incompatibilities With Ada 83

Pragma Atomic replaces Ada 83's pragma Shared. The name ‘‘Shared’’ was confusing, because the pragma was not used to mark variables as shared. 0.a

C.7 Task Identification and Attributes

C.7.1 The Package Task_Identification

C.7.2 The Package Task_Attributes

Chg839X;5.95

Annex D
(normative)

Real-Time Systems

Extensions to Ada 83

This Annex is new to Ada 9X. 0.a

D.1 Task Priorities

Extensions to Ada 83

The priority of a task is per-object and not per-type. 0.a

Priorities need not be static anymore (except for the main subprogram). 0.b

Wording Changes From Ada 83

The description of the Priority pragma has been moved to this annex. 0.c

D.2 Priority Scheduling

D.2.1 The Task Dispatching Model

D.2.2 The Standard Task Dispatching Policy

D.3 Priority Ceiling Locking

D.4 Entry Queuing Policies

D.5 Dynamic Priorities

D.6 Preemptive Abort

D.7 Tasking Restrictions

D.8 Monotonic Time

D.9 Delay Accuracy

Wording Changes From Ada 83

0.a The rules regarding a timed_entry_call with a very small positive Duration value, have been tightened to always require the check whether the rendezvous is immediately possible.

D.10 Synchronous Task Control

D.11 Asynchronous Task Control

D.12 Other Optimizations and Determinism Rules

Annex E (normative) Distributed Systems

Extensions to Ada 83

This Annex is new to Ada 9X. 0.a

E.1 Partitions

E.2 Categorization of Library Units

E.2.1 Shared Passive Library Units

E.2.2 Remote Types Library Units

An access type declared in the visible part of a remote types or remote call interface library unit is called a *remote access type*. Such a type shall be either an access-to-subprogram type or a general access type that designates a class-wide limited private type.

The following restrictions apply to the use of a remote access-to-subprogram type:

- A value of a remote access-to-subprogram type shall be converted only to another (subtype-conformant) remote access-to-subprogram type; 1
- The prefix of an Access attribute_reference that yields a value of a remote access-to-subprogram type shall statically denote a (subtype-conformant) remote subprogram. 2

The following restrictions apply to the use of a remote access-to-class-wide type:

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; the types of all the non-controlling formal parameters shall have Read and Write attributes. 3
- A value of a remote access-to-class-wide type shall be explicitly converted only to another remote access-to-class-wide type; 4
- A value of a remote access-to-class-wide type shall be dereferenced (or implicitly converted to an anonymous access type) only as part of a dispatching call where the value designates a controlling operand of the call (see E.4, ‘‘Remote Subprogram Calls’’); 5
- The Storage_Pool and Storage_Size attributes are not defined for remote access-to-class-wide types; the expected type for an allocator shall not be a remote access-to-class-wide type; a remote access-to-class-wide type shall not be an actual parameter for a generic formal access type; 6

Chg839X;5.95

E.2.3 Remote Call Interface Library Units

E.3 Consistency of a Distributed System

E.4 Remote Subprogram Calls

E.4.1 Pragma Asynchronous

E.4.2 Example of Use of a Remote Access-to-Class-Wide Type

E.5 Partition Communication Subsystem

Annex F
(normative)

Information Systems

Extensions to Ada 83

This Annex is new to Ada 9X. 0.a

F.1 Machine_Radix Attribute Definition Clause

F.2 The Package Decimal

F.3 Edited Output for Decimal Types

F.3.1 Picture String Formation

F.3.2 Edited Output Generation

F.3.3 The Package Text_IO.Editing

F.3.4 The Package Wide_Text_IO.Editing

Chg839X;5.95

Annex G
(normative)

Numerics

Extensions to Ada 83

This Annex is new to Ada 9X. 0.a

G.1 Complex Arithmetic

G.1.1 Complex Types

Wording Changes From Ada 83

The semantics of Numerics.Generic_Complex_Types differs from Generic_Complex_Types as defined in ISO/IEC CD 13813 (for Ada 83) in the following ways: 0.a

- The generic package is a child of the package defining the Argument_Error exception. 0.b
- The nongeneric equivalents export types and constants with the same names as those exported by the generic package, rather than with names unique to the package. 0.c
- Implementations are not allowed to impose an optional restriction that the generic actual parameter associated with Real be unconstrained. (In view of the ability to declare variables of subtype Real'Base in implementations of Numerics.Generic_Complex_Types, this flexibility is no longer needed.) 0.d
- The dependence of the Argument function on the sign of a zero parameter component is tied to the value of Real'Signed_Zeros. 0.e
- Conformance to accuracy requirements is conditional. 0.f

G.1.2 Complex Elementary Functions

Wording Changes From Ada 83

The semantics of Numerics.Generic_Complex_Elementary_Functions differs from Generic_Complex_Elementary_Functions as defined in ISO/IEC CD 13814 (for Ada 83) in the following ways: 0.a

- The generic package is a child unit of the package defining the Argument_Error exception. 0.b
- The proposed Generic_Complex_Elementary_Functions standard (for Ada 83) specified names for the nongeneric equivalents, if provided. Here, those nongeneric equivalents are required. 0.c
- The generic package imports an instance of Numerics.Generic_Complex_Types rather than a long list of individual types and operations exported by such an instance. 0.d
- The dependence of the imaginary component of the Sqrt and Log functions on the sign of a zero parameter component is tied to the value of Complex_Types.Real'Signed_Zeros. 0.e
- Conformance to accuracy requirements is conditional. 0.f

G.1.3 Complex Input-Output

G.1.4 The Package Wide_Text_IO.Complex_IO

G.2 Numeric Performance Requirements

Extensions to Ada 83

0.a The choice between strict and relaxed numeric performance was not available in Ada 83.

G.2.1 Model of Floating Point Arithmetic

Wording Changes From Ada 83

0.a The Ada 9X model numbers of a floating point type that are in the safe range of the type are comparable to the Ada 83 safe numbers of the type. There is no analog of the Ada 83 model numbers. The Ada 9X model numbers, when not restricted to the safe range, are an infinite set.

Inconsistencies With Ada 83

0.b Giving the model numbers the hardware radix, instead of always a radix of two, allows (in conjunction with other changes) some borderline declared types to be represented with less precision than in Ada 83 (i.e., with single precision, whereas Ada 83 would have used double precision). Because the lower precision satisfies the requirements of the model (and did so in Ada 83 as well), this change is viewed as a desirable correction of an anomaly, rather than a worrisome inconsistency. (Of course, the wider representation chosen in Ada 83 also remains eligible for selection in Ada 9X.)

0.c As an example of this phenomenon, assume that Float is represented in single precision and that a double precision type is also available. Also assume hexadecimal hardware with clean properties, for example certain IBM hardware. Then,

0.d
```
    type T is digits Float'Digits range -Float'Last .. Float'Last;
```

0.e results in T being represented in double precision in Ada 83 and in single precision in Ada 9X. The latter is intuitively correct; the former is counterintuitive. The reason why the double precision type is used in Ada 83 is that Float has model and safe numbers (in Ada 83) with 21 binary digits in their mantissas, as is required to model the hypothesized hexadecimal hardware using a binary radix; thus Float'Last, which is not a model number, is slightly outside the range of safe numbers of the single precision type, making that type ineligible for selection as the representation of T even though it provides adequate precision. In Ada 9X, Float'Last (the same value as before) is a model number and is in the safe range of Float on the hypothesized hardware, making Float eligible for the representation of T.

Extensions to Ada 83

0.f Giving the model numbers the hardware radix allows for practical implementations on decimal hardware.

Wording Changes From Ada 83

0.g The wording of the model of floating point arithmetic has been simplified to a large extent.

G.2.2 Model-Oriented Attributes of Floating Point Types

G.2.3 Model of Fixed Point Arithmetic

Inconsistencies With Ada 83

0.a Since the values of a fixed point type are now just the integer multiples of its *small*, the possibility of using extra bits available in the chosen representation for extra accuracy rather than for increasing the base range would appear to be removed, raising the possibility that some fixed point expressions will yield less accurate results than in Ada 83. However, this is partially offset by the ability of an implementation to choose a smaller default *small* than before. Of course, if it does so for a type T then T'Small will have a different value than it previously had.

0.b The accuracy requirements in the case of incompatible *smalls* are relaxed to foster wider support for non-binary *smalls*. If this relaxation is exploited for a type that was previously supported, lower accuracy could result; however, there is no particular incentive to exploit the relaxation in such a case.

Wording Changes From Ada 83

0.c The fixed point accuracy requirements are now expressed without reference to model or safe numbers, largely because the full generality of the former model was never exploited in the case of fixed point types (particularly in regard to operand perturbation). Although the new formulation in terms of perfect result sets and close result sets is still verbose, it can be seen to distill down to two cases:

- a case where the result must be the exact result, if the exact result is representable, or, if not, then either one of the adjacent values of the type (in some subcases only one of those adjacent values is allowed); 0.d
- a case where the accuracy is not specified by the language. 0.e

G.2.4 Accuracy Requirements for the Elementary Functions

Wording Changes From Ada 83

0.a The semantics of Numerics.Generic_Elementary_Functions differs from Generic_Elementary_Functions as defined in ISO/IEC DIS 11430 (for Ada 83) in the following ways related to the accuracy specified for strict mode:

- The maximum relative error bounds use the Model_Epsilon attribute instead of the Base'Epsilon attribute. 0.b
- The accuracy requirements are expressed in terms of result intervals that are model intervals. On the one hand, this facilitates the description of the required results in the presence of underflow; on the other hand, it slightly relaxes the requirements expressed in ISO/IEC DIS 11430. 0.c

G.2.5 Performance Requirements for Random Number Generation

G.2.6 Accuracy Requirements for Complex Arithmetic

Wording Changes From Ada 83

0.a The semantics of Numerics.Generic_Complex_Types and Numerics.Generic_Complex_Elementary_Functions differs from Generic_Complex_Types and Generic_Complex_Elementary_Functions as defined in ISO/IEC CDs 13813 and 13814 (for Ada 83) in ways analogous to those identified for the elementary functions in G.2.4. In addition, we do not generally specify the signs of zero results (or result components), although those proposed standards do.

Annex H
(normative)

Safety and Security

Extensions to Ada 83

This Annex is new to Ada 9X. 0.a

H.1 Pragma Normalize_Scalars

H.2 Documentation of Implementation Decisions

H.3 Reviewable Object Code

H.3.1 Pragma Reviewable

The implementation shall provide control- and data-flow information, both within each compilation unit and across the compilation units of the partition.

H.3.2 Pragma Inspection_Point

H.4 Safety and Security Restrictions

Chg839X;5.95

Annex J (normative) Obsolescent Features

Wording Changes From Ada 83

The following features have been removed from the language, rather than declared to be obsolescent: 0.a

- The package Low_Level_IO (see A.6). 0.b
- The Epsilon, Mantissa, Emax, Small, Large, Safe_Emax, Safe_Small, and Safe_Large attributes of floating point types (see A.5.3). 0.c
- The pragma Interface (see B.1). 0.d
- The pragmas System_Name, Storage_Unit, and Memory_Size (see 13.7). 0.e
- The pragma Shared (see C.6). 0.f

Implementations can continue to support the above features for upward compatibility. 0.g

J.1 Renamings of Ada 83 Library Units

J.2 Allowed Replacements of Characters

J.3 Reduced Accuracy Subtypes

Wording Changes From Ada 83

In Ada 83, a delta_constraint is called a fixed_point_constraint, and a digits_constraint is called a floating_point_constraint. We have adopted other terms because digits_constraints apply primarily to decimal fixed point types now (they apply to floating point types only as an obsolescent feature). 0.a

J.4 The Constrained Attribute

J.5 ASCII

J.6 Numeric_Error

J.7 At Clauses

Extensions to Ada 83

We now allow to define the address of an entity using an attribute_definition_clause. This is because Ada 83's at_clause is so hard to remember: programmers often tend to write ``for X'Address use...;''. 0.a

Wording Changes From Ada 83

Ada 83's address_clause is now called an at_clause to avoid confusion with the new term ``Address clause'' (that is, an attribute_definition_clause for the Address attribute). 0.b

J.7.1 Interrupt Entries

Wording Changes From Ada 83

0.a RM83-13.5.1 did not adequately address the problems associate with interrupts. This feature is now obsolescent and is replaced by the Ada 9X interrupt model as specified in the Systems Programming Annex.

J.8 Mod Clauses

Wording Changes From Ada 83

0.a Ada 83's alignment_clause is now called a mod_clause to avoid confusion with the new term ''Alignment clause'' (that is, an attribute_definition_clause for the Alignment attribute).

J.9 The Storage_Size Attribute

Annex K
(informative)

Language-Defined Attributes

Chg839X;5.95

Annex L
(informative)

Language-Defined Pragmas

Wording Changes From Ada 83

Pragmas List, Page, and Optimize are now officially defined in 2.8, ‘‘Pragmas’’. 0.a

Annex M
(informative)

Implementation-Defined Characteristics

Annex N
(informative)

Glossary

Annex P
(informative)

Syntax Summary

Syntax Cross Reference

Index

Index entries are given by paragraph number. A list of all language-defined library units may be found under Language-Defined Library Units. A list of all language-defined types may be found under Language-Defined Types.