DTIC FILE COPY

(2)

AD-A197 564

Ada Language Commentaries

Volume 1

25 July 1986 - 25 September 1987

DTIC
ELECTE
JUL 25 1988
S D
H

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

88 7 22 91

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER | 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Language Commentaries Volume 1 25 July 1986 - 25 September 1987	5. TYPE OF REPORT & PERIOD COVERED July 25 1986 - Sept. 25 1987 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Ada Joint Program Office 3D139 (1211 S. Fern, C-107) The Pentagon, Washington DC 20301-3081	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE July 1988 13. NUMBER OF PAGE 100+
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) AJPO	15. SECURITY CLASS (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)
Ada Programming Language; Ada compiler validation capability; Ada standards; Ada commentaries; Ada interpetations; ANSI/MIL-STD-1815A.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This volume contains commentaries on the Ada Standard, ANSI/MIL-STD-1815A-1983. Comments on the Standard are sent to the International Standards Organization (ISO) or the joint Program Office (AJPO) and are combined by topic into Commentaries. Based on these Commentaries, recommendations may be made to change or add to the Ada Compiler Validation Capability Test Suite.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Table of Contents

Ada-Commentary	Page
ai-00001-bi.wj	1
ai-00002-bi.wj	4
ai-00006-bi.wj	7
ai-00007-bi.wj	10
ai-00008-co.wj	22
ai-00014-bi.wj	24
ai-00015-ra.wj	27
ai-00016-bi.wj	31
ai-00018-ra.wj	34
ai-00019-bi.wj	37
ai-00020-ra.wj	40
ai-00023-bi.wj	43
ai-00024-bi.wj	46
ai-00025-bi.wj	52
ai-00026-co.wj	55
ai-00027-bi.wj	57
ai-00030-ra.wj	60
ai-00031-co.wj	63
ai-00032-ra.wj	65
ai-00034-ra.wj	67
ai-00035-ra.wj	70
ai-00037-bi.wj	72
ai-00038-ra.wj	77
ai-00039-bi.wj	79
ai-00040-ra.wj	84
ai-00045-co.wj	87
ai-00046-ra.wj	89
ai-00047-bi.wj	92
ai-00048-bi.wj	94
ai-00050-bi.wj	99
ai-00051-bi.wj	102
ai-00103-ra.wj	105
ai-00120-co.wj	107
ai-00128-bi.wj	110
ai-00132-bi.wj	113
ai-00137-ra.wj	115
ai-00138-bi.wj	117
ai-00139-ra.wj	121
ai-00143-ra.wj	125
ai-00144-bi.wj	127
ai-00145-ra.wj	130
ai-00147-ra.wj	132
ai-00148-ra.wj	134
ai-00149-bi.wj	136
ai-00150-bi.wj	139
ai-00151-bi.wj	142
ai-00153-bi.wj	144
ai-00154-ra.wj	146
ai-00155-bi.wj	150
ai-00157-bi.wj	153
ai-00163-ra.wj	156
ai-00167-co.wj	158
ai-00169-ra.wj	160
ai-00170-bi.wj	162

DTIC COPY INSPECTED 6

For
kI ☑
☐
ed ☐
ion
ion/
ity Codes
and/or
Dist Special
A-1

ai-00172-ra.wj 165
ai-00173-bi.wj 168
ai-00177-bi.wj 172
ai-00179-ra.wj 175
ai-00180-bi.wj 179
ai-00181-bi.wj 181
ai-00186-bi.wj 184
ai-00187-bi.wj 189
ai-00190-bi.wj 192
ai-00193-ra.wj 196
ai-00196-ra.wj 199
ai-00197-ra.wj 202
ai-00199-bi.wj 204
ai-00200-bi.wj 209
ai-00205-ra.wj 213
ai-00210-co.wj 216
ai-00215-bi.wj 219
ai-00219-bi.wj 221
ai-00225-bi.wj 223
ai-00226-bi.wj 225
ai-00232-co.wj 228
ai-00234-bi.wj 231
ai-00236-bi.wj 233
ai-00237-bi.wj 236
ai-00239-nb.wj 239
ai-00240-ra.wj 244
ai-00242-bi.wj 246
ai-00243-co.wj 251
ai-00244-bi.wj 253
ai-00247-co.wj 256
ai-00251-bi.wj 258
ai-00257-ra.wj 260
ai-00258-bi.wj 262
ai-00260-bi.wj 265
ai-00261-bi.wj 270
ai-00265-ra.wj 272
ai-00266-ra.wj 275
ai-00267-ra.wj 278
ai-00268-ra.wj 281
ai-00276-ra.wj 283
ai-00279-bi.wj 286
ai-00282-co.wj 289
ai-00286-bi.wj 292
ai-00287-bi.wj 297
ai-00288-nb.wj 299
ai-00289-bi.wj 302
ai-00292-bi.wj 305
ai-00293-co.wj 307
ai-00294-ra.wj 309
ai-00298-ra.wj 312
ai-00300-bi.wj 314
ai-00307-ra.wj 317
ai-00308-bi.wj 320
ai-00310-bi.wj 322
ai-00311-bi.wj 325

ai-00312-ra.wj 327
ai-00313-bi.wj 330
ai-00314-co.wj 334
ai-00316-ra.wj 336
ai-00319-co.wj 338
ai-00320-bi.wj 340
ai-00321-bi.wj 344
ai-00322-bi.wj 346
ai-00325-ra.wj 349
ai-00328-ra.wj 352
ai-00330-bi.wj 355
ai-00331-ra.wj 358
ai-00332-bi.wj 361
ai-00339-bi.wj 364
ai-00343-ra.wj 367
ai-00350-ra.wj 370
ai-00354-ra.wj 373
ai-00355-ra.wj 376
ai-00357-bi.wj 379
ai-00358-bi.wj 383
ai-00362-co.wj 387
ai-00365-bi.wj 390
ai-00370-ra.wj 392
ai-00371-bi.wj 395
ai-00376-ra.wj 398
ai-00379-co.wj 401
ai-00384-ra.wj 404
ai-00387-nb.wj 406
ai-00388-ra.wj 409
ai-00396-cr.wj 412
ai-00397-bi.wj 415
ai-00398-bi.wj 419
ai-00405-ra.wj 427
ai-00406-bi.wj 430
ai-00408-bi.wj 433
ai-00409-bi.wj 438
ai-00418-ra.wj 441
ai-00425-co.wj 445
ai-00426-bi.wj 447
ai-00444-ra.wj 450
ai-00446-bi.wj 453
ai-00449-ra.wj 456
ai-00468-cr.wj 459
ai-00471-cr.wj 462
ai-00483-cr.wj 464
ai-00486-cr.wj 466

Ada Commentary ai-00001-bi.wj downloaded on Mon May 9 10:45:02 EDT 1988

!standard 04.09 (06) 86-07-23 AI-00001/10
!standard 08.05 (04)
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-26
!status work-item 84-06-29
!status returned to committee by ADA Board 84-06-29
!status committee-approved 84-02-06
!status work-item 83-10-06
!references 83-00133, 83-00134, 83-00198, 83-00269, 83-00308, 83-00364, 83-00386, 83-00504, 83-00508
!topic Renaming and static expressions

!summary 84-03-26

If the name declared by a renaming declaration denotes a constant explicitly declared by a constant declaration having the form specified in 4.9(6), then the name can be used as a primary in a static expression.

!question 84-03-26

Is a name declared by a renaming declaration allowed in a static expression when the name denotes a constant whose name would otherwise be allowed in a static expression? In particular, does 4.9(6) refer to a name {declared} by a certain form of constant declaration or a name {denoting} a constant declared by such a constant declaration?

!recommendation 84-03-26

4.9(6) should be interpreted to mean any name DENOTING a constant explicitly declared by a constant declaration with a static subtype and initialized with a static expression.

!discussion 84-03-26

Consider the following declarations:

```
C1 : constant INTEGER := 6;
C2 : INTEGER renames C1;
```

Are expressions containing the name C2 static? 4.9(2) says that for an expression to be static, its primaries must have a certain form. 4.9(6) says that one of the allowed forms of primary is:

> "A constant explicitly declared by a constant declaration with a static subtype, and initialized with a static expression."

In deciding what this paragraph means, first note that although the intention

of the Standard is to specify different kinds of primaries, "constant" is not a syntactic form of primary [4.4]; the only relevant form here is "name." Moreover, a "constant" is an object [3.2.1(2)], not the name of an object. The issue then, is: does 4.9(6) mean a name DECLARED by a certain kind of constant declaration, or does 4.9(6) mean a name DENOTING an object declared by a certain kind of constant declaration? The second interpretation has the effect of allowing names declared by renaming declarations to be used in static expressions, since a renaming declaration does not declare an object, but a name denoting some entity [8.5(3)].

The argument in favor of interpreting 4.9(6) to mean a name declared by a constant declaration is that since 4.9 intended to specify the syntactic character of the allowed primaries of a static expression, the term "constant" in 4.9(6) was to be understood as the name declared by a certain kind of declaration. Moreover, it was the design team's intent (see comment 83-00308) to forbid the use of any name declared by a renaming declaration in a static expression.

The opposing argument is that the wording in 4.9 already allows some names declared by a renaming declaration to be used in a static expression, namely, operator symbols denoting predefined operators: 4.9(7) allows a primary of a static expression to be "a function call whose function name is an operator symbol that DENOTES [added emphasis] a predefined operator ..." Moreover, 4.9(2) says every operator in a static expression must "denote" a predefined operator. Hence, an operator declared by a renaming declaration can be used in a static expression if the renamed entity is a predefined operator. For example:

```
package P is
     type Length is new Integer;
end P;

with P;
package Length_Operations is
     subtype Length is P.Length;
     function "+" (L, R : Length) return Length renames P."+";
     function "-" (L, R : Length) return Length renames P."-";
     function "+" (L : Length) return Length renames P."+";
     function "-" (L : Length) return Length renames P."-";
end Length_Operations;

with Length_Operations; use Length_Operations;
package Application is
     Size : constant Length := 5;
     ... Size + 1 ...                    -- static expression
     Area : Length := Size * Size;       -- illegal; * not visible
```

The expression Size+1 is static since the "+" operator denotes the predefined function "+" that is implicitly declared in P.

The conclusion of the Committee is:

- 4.9(6) can be interpreted in two ways;
- the evidence of 4.9(2) and 4.9(7) (i.e., the use of "denotes") implies that names of constants should be allowed in static expressions based on what the names denote.

Ada Commentary ai-00002-bi.wj downloaded on Mon May 9 10:45:03 EDT 1988

!standard 08.03 (17) 86-12-01 AI-00002/07
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status WG14/ADA Board approved (provisional) 84-11-27
!status WG14-approved (provisional) 84-11-27
!status board-approved (provisional) 84-11-26
!status committee-approved 84-06-28
!status work-item 83-10-06
!reference AI-00330, 83-00138, 83-00140, 83-00305, 83-00590, 83-00597
!topic Deriving homographs for an enumeration literal and a function

!summary 84-09-10

It is possible to derive both an enumeration literal and a user-defined function that is a homograph of the literal. In such cases, the enumeration literal is hidden by the user-defined function.

!question 84-09-10

Consider a user-defined function that is a homograph of an enumeration literal. If a derived type declaration derives both the function and the literal, which of these derived entities, if any, is visible?

!recommendation 84-09-10

If the implicit declaration of a derived enumeration literal is a homograph of the implicit declaration of a derived subprogram and these declarations occur immediately within the same declarative region, then the derived enumeration literal (like a predefined operation) is hidden by the other homograph. The derived enumeration literal is hidden within the entire scope of the derived subprogram's declaration.

!discussion 86-09-13

8.3(17) says:

> Two declarations that occur immediately within the same declarative region must not be homographs, unless either or both of the following requirements are met: (a) exactly one of them is the implicit declaration of a predefined operation; (b) exactly one of them is the implicit declaration of a derived subprogram. In such cases, a predefined operation is always hidden by the other homograph; a derived subprogram hides a predefined operation, but is hidden by any other homograph.

Consider the following example in light of 8.3(17):

```
package P is
   type T is (RED);
   type NT is private;
```

```
      function RED return NT;
   private
      type NT is new T;
      type NNT is new NT;
      -- literal RED and function RED declared here
   end P;
```

For the full declaration of NT, the derived enumeration literal is hidden by the explicit function RED, but is still derivable according to 3.4(6). Thus for the declaration of NNT, we get two implicit declarations of homographs. This case is not clearly covered by 8.3(17), because an enumeration literal is not explicitly defined to be a predefined operation in 3.5.5(15) and it is not a derived subprogram in the sense of 3.4(11).

If enumeration literal RED is not considered a predefined operation with respect to 8.3(17), then the declaration of NNT is illegal because it requires the illegal declaration of two homographs. But since an enumeration literal was intended to have the status of a predefined operation for an enumeration type, at least with respect to 8.3(17), the declaration of NNT is legal and the visible entity denoted by RED is the user-defined function, not the enumeration literal.

Ada Commentary ai-00006-bi.wj downloaded on Mon May 9 10:50:02 EDT 1988

!standard 05.05 (06) 86-07-23 AI-00006/05
!class binding interpretation 83-10-18
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status work-item 83-10-07
!references 83-00077, 83-00078, 83-00079, 83-00080, 83-00081, 83-00199
!topic The subtype of a loop parameter

!summary 84-03-26

The subtype of a loop parameter is determined by the discrete range in the loop parameter specification. Therefore, the loop parameter has a static subtype if the discrete range is static.

!question 84-07-13

Is the subtype of a loop parameter determined by the discrete range used in the loop statement, or is the subtype always equivalent to the base type?

!recommendation 83-10-07

The intent was to have the loop parameter subtype be determined by the discrete range of the loop parameter specification. In particular, if the discrete range is static, the loop parameter has a static subtype.

!discussion 84-03-14

5.5(6) says, "The loop parameter is an object whose type is the base type of the discrete range." This wording does not explicitly define the subtype of a loop parameter. The only time this matters is when the loop parameter is used in a case statement:

```
for J in 1..2 loop
     case J is
          when 1 => ... ;
          when 2 => ... ;
     end case;
```

If J is considered to have the static subtype, INTEGER range 1..2, then the case statement is legal; otherwise, an OTHERS choice is required to ensure that all values of the base type are represented by the set of choices [5.4(4)]. The July 1980 RM suggested that the subtype was 1..2, and there was no intent to change from this position.

The validation tests currently take the position that the above case statement is legal, and three compilers have been validated against these tests. At least three organizations, however, have protested these tests, insisting that the above case statement is illegal.

Since the validation tests agree with the intent, the recommended interpretation is that the subtype of a loop parameter be determined by the discrete range; the above case statement is legal.

Ada Commentary ai-00007-bi.wj downloaded on Mon May 9 10:50:04 EDT 1988

!standard 03.07.02 (05) 86-12-04 AI-00007/19
!standard 03.08.01 (04)
!class binding interpretation 86-05-12
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (6-0-0) 86-09-11 (pending editorial review)
!status work-item 86-09-10
!status failed letter ballot (1-9-3) 86-09
!status committee-approved (10-0-0) 86-05-12 (pending letter ballot)
!status committee-approved (8-0-0) 86-02-20 (pending editorial review)
!status work-item 85-09-04
!status committee-approved (7-1-1) 85-05-18
!status work-item 84-11-27
!status ADA Board referred back to committee 84-11-26
!status WG14 referred back to committee 84-11-27
!status committee-approved 84-06-28
!status work-item 83-10-07
!references AI-00358, AI-00014, 83-00082, 83-00083, 83-00084, 83-00085,
83-00086, 83-00110, 83-00111, 83-00112, 83-00113, 83-00114,
83-00271, 83-00273, 83-00274, 83-00282, 83-00469, 83-00498,
83-00526, 83-00614, 83-00645, 83-00646, 83-00647, 83-00664,
83-00662, 83-00666, 83-00689, 83-00693, 83-00698, 83-00727,
83-00788, 83-00796, 83-00798, 83-00801
!topic Discriminant compatibility for incomplete, private, and access types

!summary 86-12-04

When a subtype indication with a discriminant constraint is elaborated, 3.3.2(6-8) requires that the compatibility check defined in 3.7.2(5) be performed. The check has two parts: first, check that the value of each discriminant belongs to the corresponding discriminant subtype, and second, if a discriminant is used in a subcomponent constraint, check the constraint for compatibility with the subcomponent's type. If a discriminant constraint is applied to a private type or to an incomplete type before its complete declaration, the second part of the check cannot be performed when the subtype indication is elaborated because no subcomponent declarations exist.

The recommended interpretation of 3.7.2(5) in this case is that the check for subcomponents be deferred and be performed no later than the end of the declaration that allows the deferred check to be completely performed, except when an incomplete type is declared in the private part of a package and its full declaration is given in the package body; in this case, a discriminant constraint is not allowed for the type prior to the end of the package specification.

If a discriminant constraint is given for an access type, the constraint applies to the designated type. The second part of the compatibility check is optional in this case (even if the designated type is completely declared, e.g., even if the constraint occurs in an object declaration or allocator).

When the initial values of discriminants are given by the evaluation of default expressions, the corresponding constraint is checked for compatibility.

!question 84-09-03

When a discriminant constraint is applied to a private type before its full declaration or to an incomplete type before its full declaration, 3.3.2(8) and 3.7.2(5) require that if a discriminant is used to constrain a subcomponent, its value must be checked for compatibility with the subcomponent's type. However, prior to the full declaration, no subcomponents exist (other than the discriminants themselves). When should the required check be performed, if ever? Should such subtype indications be considered illegal?

!recommendation 86-09-13

A discriminant constraint is not allowed in a subtype indication given in the private part of a package if the type mark denotes an incomplete type and the incomplete type's full declaration is given in the package body.

A discriminant constraint is compatible with a type that has discriminants if and only if two conditions are met. First, each discriminant value belongs to the subtype of the corresponding discriminant. Second, if the subtype defined by the constraint has components whose component subtype definitions depend on a discriminant, the corresponding discriminant value is substituted for the discriminant in each such component subtype definition and the compatibility of the resulting subtype indication is checked; if the type has not been completely declared, the substitution and check of discriminant values for compatibility is performed no later than the end of the type's complete declaration.

A discriminant constraint is compatible with an access type if and only if two conditions are met. First, each discriminant value belongs to the subtype of the corresponding discriminant of the designated type. Second, if a full type declaration has been given for the designated type and the subtype defined by imposing the constraint on the designated type has components whose component subtype definitions depend on a discriminant, the corresponding discriminant value may (but need not) be substituted for the discriminant in each such component subtype definition; if this substitution is done, the compatibility of the resulting subtype indication is checked.

When the initial values of discriminants are given by the evaluation of default expressions, the corresponding constraint is checked for compatibility.

(The complete declaration of an incomplete or private type is its full declaration if the full declaration declares a scalar type or an access type. If the full declaration declares a type with components, the full declaration is the complete declaration only if the type of each component has been completely declared; otherwise, the type is completely declared by the last complete declaration of a component's type. This meaning of "complete declaration" is also used in AI-00260.)

!discussion 86-12-04

The discussion is divided into the following sections:

- presentation of the problem
- presentation of possible solutions
- discussion of the recommended solution
  - deferring the check
  - records with variant parts
  - treatment of access types
  - compatible default discriminant values
- summary

THE PROBLEM

Consider either of the following two analogous cases:

```
type T (D : INTEGER) is private;  |    type T (D : INTEGER);
subtype U is T(D => -1);          |    type U is access T(-1);
                          N : INTEGER := -3;
        -- declarations whose elaboration changes the value of N
private                                 |
                      type T (D : INTEGER) is
                          record
                              S : STRING (D .. N);
                              -- exception if D < N and D < 1
                          end record;
```

Section 3.7.2(5) says, in part:

> for each subcomponent whose component subtype [definition] depends on a discriminant, the discriminant value is substituted for the discriminant in this component subtype [definition] and the compatibility of the resulting subtype indication is checked.

The wording suggests that when the declarations of U above are elaborated, CONSTRAINT_ERROR should be raised if -1 .. N is not a compatible index constraint for STRING. However, the required check cannot be done correctly without first elaborating the intervening declarations (since, as indicated, the intervening declarations may change the value of N); the full type definition must also be elaborated before the check can be made (to evaluate the upper bound of S). In particular, if the value of the upper bound of S is -3, no exception need be raised for T(-1) since a null range is specified. If the value is greater than -1, CONSTRAINT_ERROR should be raised. But at

the time T(-1) is elaborated (i.e., prior to elaborating later declarations), it is not possible to tell whether CONSTRAINT_ERROR should be raised. In short, the Standard requires a check that cannot be performed, in general, at the required point.

The same problem occurs when the type mark is an access type:

```
type AC_T is access T;      -- using either definition of T
subtype V is AC_T(-1);      -- exception?
```

The wording in 3.7.2(5) applies to the subtype indication given in V's declaration, so the compatibility of -1 for T must be checked when V's subtype indication is elaborated. But just as for U's declaration, the information needed to perform the check completely is not yet available.

The check required by 3.7.2(5) has two parts: 1) a check that the discriminant value belongs to the discriminant subtype, and 2) an "additional" check on subtypes of components that depend on the discriminant. The first check can always be performed when the subtype indication is elaborated since it only uses the subtype of each discriminant. Only the second check (the "subcomponent check") causes difficulty, and this difficulty only occurs when the type being constrained is not yet completely declared, i.e., since all its subcomponents are not yet declared, the information needed to perform the second part of the check is not available when the subtype indication is elaborated.

POSSIBLE SOLUTIONS

Several possible solutions to this problem have been considered:

(1) Make it illegal to constrain any private or incomplete type prior to its full declaration since the check required by 3.7.2(5) cannot be fully performed when the subtype indication is elaborated. This resolution is more drastic than the recommended one and removes potentially useful capabilities from the language. For example, it is useful to be able to declare a subtype of a private type in the visible part of the package that declares the type.

(2) Perform the first part of the check for a private or incomplete type when the subtype indication is elaborated, but only make the subcomponent check when all subcomponents of the constrained type have been declared. This solution requires that each discriminant constraint (for an incomplete or private type) be evaluated and the discriminant values saved until the complete declaration is elaborated. (The declaration that completes the declaration of all subcomponents of the constrained type is said to be the "complete" declaration for the constrained type.) The implications of this solution are discussed later.

(3) Perform the first part of the check for a private or incomplete type when the subtype indication is elaborated, but don't perform the second part of the check. Instead, perform the subcomponent compatibility check when an attempt is made to create an object. Since no objects can be created until all subcomponents are declared, this solution ensures that the compatibility

check can always be performed. It implies that some subtype declarations will not raise any exception even though every attempt to create an object having that subtype will raise an exception. For example:

```
package P is
     type ACC_STR is access STRING;
     type PRIV (D, E : INTEGER) is private;
     type NON_PRIV (D, E : INTEGER) is
          record
               S : STRING (D .. 3);
               T : ACC_STR (E .. 3);
          end record;
     subtype PRIV_S is PRIV (-1, 1);          -- no CONSTRAINT_ERROR
                                              -- no subcomponents yet
     subtype NON_PRIV_S is NON_PRIV(-1, 1);   -- CONSTRAINT_ERROR
                                         -- because of subcomponent check
     subtype PRIV_T is PRIV (1, -1);          -- no CONSTRAINT_ERROR
                                              -- no subcomponents yet
     subtype NON_PRIV_T is NON_PRIV(1, -1);   -- CONSTRAINT_ERROR
                                         -- because of subcomponent check
     type ACC_PRIV is access PRIV;
private
     type PRIV (D, E : INTEGER) is
          record
               S : STRING (D .. 3);
               T : ACC_STR (E .. 3);
          end record;
end P;
use P;
subtype P1 is PRIV_S;         -- no check since PRIV_S is private type
subtype P2 is PRIV (-1, 1);   -- no check since PRIV is private type
subtype P3 is PRIV_T;         -- no check since PRIV_T is private type
subtype P4 is PRIV (1, -1);   -- no check since PRIV is private type

X1 : PRIV_S;                  -- attempt to create constrained object
X2 : PRIV (1, -1);            -- attempt to create constrained object
X3 : ACC_PRIV (1, -1);        -- no designated object created; no exception
X4 : ACC_PRIV (1, -1)
       := new PRIV (1, -1); -- CONSTRAINT_ERROR for allocator
```

CONSTRAINT_ERROR is raised for the declarations of X1 and X2 because the subcomponent check is performed and is not successful. In the declaration of X3, no exception is raised when access object X3 is created because each of the discriminant values specified in the discriminant constraint belong to the corresponding discriminant subtypes for the designated type, PRIV. Since the designated type is a private type, no subcomponent check is made. An access value is created and given the default value, null. In the declaration of X4, however, evaluation of the allocator, new PRIV (1, -1), raises CONSTRAINT_ERROR because the value -1 is not compatible with its use in constraining a subcomponent of the designated object.

The main disadvantage of this solution is that it allows a subtype indication

to be elaborated without raising any exception even though later use of the subtype causes an exception to be raised (e.g., see the declaration of X1).

4) Make subcomponent checks unnecessary by ensuring that if a discriminant value belongs to the discriminant's subtype, it will be compatible with its use to constrain any subcomponent. This approach applies a kind of "contract model" to types that have discriminants. For example,

```
type REC (D : INTEGER) is
     record
          S : STRING (D .. 1);
     end record;
```

This declaration would raise CONSTRAINT_ERROR because if D has a value less than one, this value would be incompatible with D's use in constraining component S. This idea extends to variant parts:

```
type REC2 (D : INTEGER) is
     record
          case D is
               when POSITIVE =>
                    S : STRING (D .. 1);
               when others   =>
                    null;
          end case;
     end record;
```

No exception would be raised for the above declaration because all possible values of D in the declaration of component S are compatible.

This alternative has been rejected primarily because it is a significant change to the language. Certain declarations that are currently acceptable and potentially useful would no longer be allowed. This proposal has been presented in comments 83-00662 and 83-00693.

DISCUSSION

Of the various alternatives, making the subcomponent check no later than the point of the complete declaration seems the most acceptable, except when the complete declaration occurs in a different unit, namely, a package body and except when an access type is being constrained. We will first discuss the implications of deferring the subcomponent check until complete declarations are elaborated, and then we will discuss the issues involving access types and certain incomplete types.

DEFERRING THE CHECK

In the examples given so far, it has been sufficient to defer making the subcomponent check until the full declaration is elaborated. More complex examples are possible, however, in which the full declaration itself contains subtype indications whose compatibility cannot be checked until later declarations are elaborated.

Two choices were considered in deciding when the deferred check must be performed:

1) perform a deferred check as soon as possible, i.e., when each component subtype definition using a discriminant is elaborated; and

2) perform a deferred check when the type being constrained is "completely" declared, i.e., when all subcomponents have been declared.

The recommended solution allows deferred checks to be performed in accordance with choice 1 and requires that deferred checks be performed no later than in accordance with choice 2. To see the differences between the choices and the consequences of the recommendation, consider the following example:

```
package P is
     subtype Int7 is Integer range 1..7;
     subtype Int6 is Integer range 1..6;
     subtype Int5 is Integer range 1..5;

     type T_Int7 (D7 : Int7) is private;
     type T_Int6 (D6 : Int6) is private;
     type T_Int5 (D5 : Int5) is private;

     subtype T_Cons is T_Int7(6);              -- (1)

private

     type T_Int7 (D7 : Int7) is
          record
               C76 : T_Int6(D7);             -- (2)
          end record;

     type T_Int6 (D6 : Int6) is
          record
               C65 : T_Int5(D6);             -- (3)
               CF  : T_Int5(Func);
          end record;                        -- (4)

     type T_Int5 (D5 : Int5) is
          record
               CF  : String (Func .. D5);    -- (5)
          end record;                        -- (6)
end P;
```

In this example, the full declaration of T_Int7 is not the complete declaration of T_Int7 because the declaration of T_Int6 is not yet complete. Similarly, T_Int6's full declaration is not the complete declaration for T_Int6 or for T_Int7 because of the use of T_Int5. Finally, T_Int5's full declaration is a complete declaration for T_Int5 and so completes the declaration of T_Int6 and T_Int7.

CONSTRAINT_ERROR would be raised at point (1) (in accordance with the first compatibility check) if the discriminant's value were not in the range 1..7. Since it is in the required range, no exception is raised. Subsequent checks are deferred compatibility checks that depend on the use of the discriminant in T_Int7's full type declaration. The declaration at (2) requires yet another deferred check, since T_Int6 has not yet been fully (or completely) declared, and similarly the declarations at (3) require deferred checks. At the end of T_Int5's declaration, T_Int5, T_Int6, and T_Int7 have been completely declared and no additional deferred checks are required. The recommendation requires that all deferred checks for types whose declaration is completed by T_Int5's declaration be performed no later than point (6). Since the value 6 is not compatible with D6's use at point 3, the checks deferred from point (2) and point (1) will require that CONSTRAINT_ERROR be raised. The exception can be raised at any point between point (3) (the first point at which a deferred check can fail) and point (6) (the point by which all deferred checks in this example must have been completed). (Note that if the declaration at (3) had been

```
C65 : T_Int5(6);
```

CONSTRAINT_ERROR would have been raised immediately, since the first check required by 3.7.2(5) would fail.)

From an implementation viewpoint, it is reasonable to require that the deferred checks be performed no later than point (6) since implementations must keep track of when types are completely declared (in order to enforce 7.4.1(4)); keeping track of the deferred checks as well is likely to be only a slight additional burden. On the other hand, some implementations might find it easier to apply deferred checks on a step by step basis, e.g., checking at point (2) that the value 6 is compatible with D6's constraint and then checking at point (3) that 6 is compatible with D5's constraint. This approach is allowed by the recommendation. Requiring the deferred checks to be performed at any specific point earlier than point (6) might be inconvenient for some implementations, but being more specific about where the checks are performed is of only marginal value to a programmer, whose program is, in any case, in error.

RECORDS WITH VARIANT PARTS

In the following example, a variant part contains a component subtype definition that depends on a discriminant. The recommended interpretation requires that the compatibility of the component's discriminant constraint be checked only if the component exists in the subtype:

```
package P is
    type REC1 (INT : INTEGER) is private;
    subtype SREC1 is REC1(-2);              -- deferred check
private
    type REC2 (INT : INTEGER; POS : POSITIVE) is
        record
            C1 : STRING (INT .. 3);
        end record;
```

```
        type REC1 (INT : INTEGER) is
            record
                case INT is
                    when -1..INTEGER'LAST =>
                        C2 : REC2 (INT, 1);
                    when others =>
                        C3 : INTEGER;
                end case;
            end record;
    end P;
```

Since component C2 does not exist for subtype REC1(-2) and since the recommended interpretation says the additional compatibility check is only performed for components of the SUBTYPE being checked, no compatibility check is performed for REC2(-2, 1); therefore, no exception is raised when REC1's complete declaration is elaborated. Of course, if subtype SREC1 had been specified as REC1(-1), component C2 would exist and CONSTRAINT_ERROR would be raised no later than the end of REC1's complete declaration. (See AI-00358 for further discussion of compatibility checks for components that depend on a discriminant.)

TREATMENT OF ACCESS TYPES

The recommended interpretation does not require a subcomponent compatibility check when a discriminant constraint is applied to an access type. Such checks can be difficult or impossible to perform correctly and they are not necessary (i.e., failure to perform the check does not allow an invalid object to be created).

First, let's consider why it is safe to eliminate the subcomponent check. Consider:

```
    type ACC_STR is access STRING;
    type VSTR (FIRST, LAST : INTEGER) is
         record
              DATA : ACC_STR (FIRST .. LAST);
         end record;
    X : VSTR (-1, -1);                  -- optional exception
```

CONSTRAINT_ERROR will be raised if a subcomponent check is performed for X.DATA's designated type, since -1 is not an allowed lower bound value for non-null STRINGs. If no such check is performed, then an exception will be raised later:

```
    X := (-1, -1, new STRING (-1..-1));  -- CONSTRAINT_ERROR raised (1)
    X := (1, 3, new STRING (1..3));      -- CONSTRAINT_ERROR raised (2)
    X := (-1, -1, new STRING'("abc"));   -- CONSTRAINT_ERROR raised (3)
```

In case (1), an exception is raised by the allocator, since -1 is not an allowed lower bound. In case (2), the aggregate produces an allowed value of type VSTR, but an exception is raised by the assignment since X's discriminant constraint is not satisfied. In case (3), CONSTRAINT_ERROR is

raised because the bounds of the designated object are not the same as the discriminant values. Now consider a somewhat more complex example:

```
type ACC_VSTR is access VSTR;
Y : ACC_VSTR (-1, -1);          -- no exception need be raised
```

An allocator creating a value to be assigned to Y need not raise an exception either:

```
Y := new VSTR (-1, -1);
```

This is acceptable since Y.DATA = null; no invalid object has been created. On the other hand, an implementation could raise CONSTRAINT_ERROR for the allocator if it performs the subcomponent check for the DATA component.

Allowing the subcomponent check to be eliminated simplifies an implementation considerably. For example, consider:

```
type R1 (D1 : INTEGER);
type R2 (D2 : INTEGER);
type R3 (D3 : POSITIVE);

type ACC_R1 is access R1;
type ACC_R2 is access R2;
type ACC_R3 is access R3;

type R1 (D1 : INTEGER) is
     record
          C1 : ACC_R2 (D1);
     end record;

X1 : R1 (-1);           -- can't do complete subcomponent check

type R2 (D2 : INTEGER) is
     record
          C2 : ACC_R3 (D2);
     end record;

type R3 (D3 : POSITIVE) is
     record
          C3 : ACC_R1 (D3);
     end record;
```

The declaration of X1 would be illegal if R1 were an incomplete type or if C1's type were a private type prior to its complete declaration. Since C1 has an access type, X1's declaration is legal. A complete subcomponent check can't be performed when X1 is declared since R2 has not yet been completely declared. Nonetheless, the object denoted by X1 can be created. One could even create a function (by generic instantiation) that could be called with X1 as an actual parameter. In short, requiring a deferred check of subcomponent compatibility does not seem useful in this case.

In addition, performing such a subcomponent check is complex. Suppose the following declaration appears after R3's full declaration and an implementation attempts to perform the subcomponent check:

```
A1 : ACC_R1 (10);
```

When performing the check for A1, an implementation must be careful not to get into a loop, since both A1 and A1.C1.C2.C3 have the same access type. The complexity of avoiding a loop is increased when one considers that any of the designated types could be variant records, i.e., the choice of which subcomponents to check could depend on which subcomponents are present for particular discriminant values (see AI-00358). It seems pointless to require such complexity for access types, since such checks cannot always be performed when an object is declared (see the example with X1) and since such checks are never needed to prevent the creation of an invalid value.

Only the subcomponent check can be omitted:

```
A1 : ACC_R3 (-1);        -- raises CONSTRAINT_ERROR
```

CONSTRAINT_ERROR must be raised since -1 does not belong to R3's discriminant subtype. No subcomponent compatibility check is needed.

Another advantage of not requiring a subcomponent check for constrained access types is connected with the use of incomplete types. If an incomplete type is declared in the private part of a package, its full declaration can be given in the package body. If the incomplete type has discriminants and is constrained, any subcomponent check could not be performed until the body was compiled:

```
package P is
     type T (D : INTEGER) is private;
     ...
private
     type THE_REAL_T (D : INTEGER);
     type ACC_REAL_T is access THE_REAL_T;
     type T (D : INTEGER) is
          record
               S : ACC_REAL_T(D);
          end record;
     OBJ : ACC_REAL_T(-1);                    -- no subcomponent check
     type R1 is access THE_REAL_T (-1);       -- not allowed
end P;
```

The object declaration is allowed, but since the designated type has not been completely declared, no subcomponent check can be performed (and none is required). R1's type declaration is not allowed, since the subcomponent check would have to be deferred until the package body is compiled. Requiring such deferred checks is too much of an implementation burden considering the potential benefits (which are practically nil).

COMPATIBLE DEFAULT DISCRIMINANT VALUES

Since discriminant_constraint is a syntactic term, and since 3.7.2(5) specifies how to check discriminant_constraints for compatibility, the definition in 3.7.2(5) does not, technically speaking, apply when discriminants are defined by default, since no discriminant_constraint then appears. Obviously, the intent was to use the same compatibility definition even when discriminant values are given implicitly. The recommended interpretation therefore says that default discriminant values must be checked for compatibility; AI-00014, AI-00358, and AI-00449 discuss when default discriminant expressions are to be evaluated.

SUMMARY

Determining the correct treatment of subcomponent compatibility checks has required a careful analysis of several alternatives. The recommended alternative ensures that invalid objects cannot be created. It imposes some implementation burden (by requiring that certain checks be deferred), but also eases the implementation burden by allowing such checks to be omitted for constrained access types; current implementations that perform the checks need not change. In addition, making certain subtype indications illegal removes an implementation burden without limiting the effective use of incomplete types whose full declaration is given in a separate package body.

The recommended interpretation is unlikely to make any existing programs illegal. If an implementation takes full advantage of the option of not performing subcomponent checks for constrained access types, the point at which an exception is raised will change in some programs, but since such programs are in any case incorrect, this should not have any practical consequences.

Ada Commentary ai-00008-co.wj downloaded on Mon May 9 11:45:02 EDT 1988

!standard 02.04.02 (04) 86-07-23 AI-00008/05
!class confirmation 84-03-16
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status work-item 83-10-30
!references 83-00093
!topic Negative exponents in based notation

!summary 84-03-16

The exponent of a based literal must not have a minus sign if the based literal is an integer literal.

!question 84-07-13

Is 8#777#E-3 a legal integer literal even though it contains a minus sign in its exponent?

!response 84-01-15

The final sentence in 2.4.1(4) mentions integer literals, and so applies to based literals, since 2.4(1) defines the form of an integer literal, and this form includes based literals.

Ada Commentary ai-00014-bi.wj downloaded on Mon May 9 11:45:04 EDT 1988

!standard 03.07.02 (08) 87-06-18 AI-00014/10
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (8-0-0) 86-09-10 (pending editorial review)
!status work-item 86-09-10
!status failed letter ballot (1-9-3) 86-09
!status committee-approved (7-2-0) 86-05-12 (pending letter ballot)
!status work-item 83-10-10
!references 83-00074, 83-00108, 83-00392, 83-00784, 83-00798
!topic Evaluating default discriminant expressions

!summary 86-09-13

Default discriminant expressions are not evaluated when an explicit initialization expression is provided in an object declaration or a component declaration.

!question 86-01-28

3.7.2(8) says that in the absence of a discriminant constraint in an object declaration that declares a variable, the discriminant values of the object are defined by the default expressions. Does this mean the default expressions are evaluated even when an explicit initialization expression is given in the object declaration?

!recommendation 86-09-13

In the absence of a discriminant constraint in an object declaration, discriminant values for the object are defined either by the initial value given in the declaration or, in the absence of an initial value, by the default expressions.

In the absence of a discriminant constraint in a component declaration, discriminant values for the component are defined either by the initial value given in the component declaration, an initial value given for the enclosing record type, or in the absence of any such initial values, the default expressions for the discriminants.

!discussion 86-09-13

3.2.1(6) says:

> If the object declaration includes an explicit initialization, the initial value is obtained by evaluating the corresponding expression. Otherwise any implicit initial values for the object or for its subcomponents are evaluated.

Similarly, for components of objects, 3.2.1(14) says:

> In the case of a component that is itself a composite object and whose value is defined neither by an explicit initialization nor by a default expression, any implicit initial values for components of the composite object are defined by the same rules as for a declared object.

On the other hand, 3.7.2(8) says:

> For a variable declared by an object declaration, the subtype indication of the corresponding object declaration must impose a discriminant constraint unless default expressions exist for the discriminants; the discriminant values are defined either by the constraint or, in its absence, by the default expressions.

The phrase "the discriminant values are defined either by the constraint or, in its absence, by the default expressions" is incorrect since the discriminant values are not defined by the default expressions when an explicit initial value is given. Moreover, 3.2.1(6 and 14) state the intended rule: if an object or component declaration has an explicit initial value, default discriminant expressions are not evaluated.

Ada Commentary ai-00015-ra.wj downloaded on Mon May 9 11:50:01 EDT 1988

| !standard 04.01.04 (03) 86-12-01 AI-00015/12
| !standard 13.07.02 (06)
| !class ramification 85-01-28
| !status approved by WG9/AJPO 86-11-26
| !status approved by Director, AJPO 86-11-26
| !status approved by WG9/Ada Board 86-11-18
!status approved by WG9 86-05-09 (provisional)
!status committee-approved (10-0-0) 85-11-22
!status work-item 85-01-28
!status received 84-01-15
!references AI-000157, 83-00070, 83-00272, 83-00071, 83-00178, 83-00243, 83-00418, 83-00593, 83-00601, 83-00679
!topic When the prefix of 'ADDRESS contains a function name

!summary 85-12-27

The name of an attribute designator can be taken into account when deciding whether the prefix of an attribute is a name or a function call, but the attribute designator cannot be considered when resolving identifiers that are used in the prefix. In particular, the fact that the prefix of 'ADDRESS (as well as the prefix of 'SIZE, 'CONSTRAINED, and 'STORAGE_SIZE) can be an object but not a function call does not affect the resolution of a name that occurs in the prefix.

!question 85-10-31

4.1.4(3) says:

> 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.

Now consider this example:

```
type ACC_STRING is access STRING;
function F (X : INTEGER) return ACC_STRING;     -- F#1
function F return ACC_STRING;                   -- F#2
...
F(3)'ADDRESS          -- ambiguous?
```

The prefix of 'ADDRESS is not allowed to be a function call (since 13.7.2(2) says the prefix must be an object), so the prefix cannot be considered an invocation of F#1. Does this mean that the F in F(3) is unambiguously resolved as F#2 (so the expression yields the address of the third component of the object designated by F's value)?

!response 86-09-29

8.3(1) says:

> The meaning of the occurrence of an identifier at a given place in the text is defined by the visibility rules and also, in the

case of overloaded declarations, by the overloading rules.

(The prefix F(3) does not have a meaning in the sense of 8.3(1) since it is not an identifier, but we can understand 4.1.4(3)'s use of the phrase "meaning of the prefix" as entailing "the meaning of the identifiers in the prefix".)

8.7(2) says:

> For overloaded entities, overload resolution determines the actual meaning that an occurrence of an identifier has, whenever the visibility rules have determined that more than one meaning is acceptable at the place of this occurrence. ...

When more than one declaration of an identifier is visible, overloading resolution is required. 8.7(7) says:

> When considering possible interpretations of a complete context, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form described below [in 8.7(8-13)].

In particular, note that use of "the syntax rules" is required in deciding what the interpretation (i.e., meaning) of an identifier is.

Now let's consider the effect of these rules on some examples that are simpler than the one posed in the question. First, consider:

```
type ACC_STRING is access STRING;
procedure G (X : INTEGER) is ... end;
function  G (X : INTEGER) return ACC_STRING is ... end;

        ... G(3)'LAST ...       -- legal?
        ... G(3)'SIZE ...       -- legal?
```

First, it is clear that the visibility rules alone give two "meanings" for G, so overloading resolution must be attempted. Overloading resolution takes the syntax rules into account. In this case, the syntax rules for uses of the prime (') require that G(3) be considered a prefix (of an attribute), and a prefix must be either a name or a function call. Since the syntax rules forbid an interpretation of G(3) as a procedure call, G is uniquely determined to denote function G. Note that in making this determination, we have not used any information about the attribute designator nor any information stemming from the use of G(3) as the prefix of an attribute; we have only used the fact that G(3) must be a prefix.

In short, overloading resolution of the prefix of an attribute can take into account the fact that only a name or a function call is allowed, but it cannot take into account the nature of the attribute designator. If the identifiers in the prefix are unambiguous given the fact that they are being used in a prefix, the "meaning" of the identifiers has been determined. At this point, the attribute designator can be used to decide whether the prefix

is to be considered a function call or not. (Since the meaning of identifiers in the prefix has been determined, treating the prefix as a function call or name cannot further affect the "meaning" of the prefix.)

For example, consider:

```
function H return ACC_STRING is ... end;

... H'LAST ...          -- legal?
... H'ADDRESS ...       -- legal?
```

Here H unambiguously denotes a specific function, so the "meaning" of H is unambiguous, but consideration of the prefix alone is insufficient for deciding whether H should be treated as a name or a function call. However, once the meaning has been uniquely determined, 4.1.4(3) does not forbid using the attribute designator to decide how the prefix should be interpreted. A prefix for the attribute LAST must be appropriate for an array type or the prefix must denote a constrained array subtype (see A(21)). Since H does not denote a subtype, it must be appropriate for an array type, i.e., the prefix must have either an array type or an access type whose designated type is an array type. This means the prefix cannot be the name of a function, but it can be a function call, so H'LAST is unambiguous (H is called). Similarly, for H'ADDRESS, 13.7.2(6) requires that:

> if the prefix is the name of a function, the attribute is understood to be an attribute of the function (not of the result of calling the function).

Since the prefix in this case cannot be a function call, there is no ambiguity in deciding how to treat the prefix (H is not called), so H'ADDRESS is legal.

This example shows how the attribute designator can be taken into account when deciding whether the prefix is to be considered a name or a function call, even though the designator cannot be used to help resolve the "meaning" of names occurring in the prefix.

Now let's use this approach in considering the legality of F(3)'ADDRESS in the original question. F in the prefix has two meanings according to the visibility rules (i.e., F can denote either F#1 or F#2). If we try to resolve the meanings by taking into account the fact that F(3) is a prefix, we find that F(3) can be considered either a function call or a name, depending on which declaration F is considered to denote. This means F(3) is ambiguous even after taking into account the fact that it is a prefix. Since the prefix is ambiguous, F(3)'ADDRESS is ambiguous; (4.1.4(3) forbids using the attribute designator to help resolve the "meaning" of identifiers in the prefix). The fact that F(3)'ADDRESS is only legal if F is considered to denote F#2 is irrelevant because 4.1.4(3) does not allow us to use this information.

The rule in 4.1.4(3) is not equivalent to saying that the prefix of an attribute is a "complete context." If the prefix were considered a complete

context, we could use any rules that determine the syntactic structure of the prefix to help resolve the prefix. In particular, we could use the rules specified for each attribute designator to decide whether a function call is allowed as a prefix. In the original example, such rules would show that F(3) cannot be considered a function call, so F(3)'ADDRESS would be unambiguous.

Ada Commentary ai-00016-bi.wj downloaded on Mon May 9 11:50:02 EDT 1988

!standard 04.01.03 (15) 86-08-05 AI-00016/10
!standard 04.01.03 (18)
!class binding interpretation 85-11-22
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9+1-0-2) 86-02-20
!status committee-approved (4-1-3) 85-11-22 (pending letter ballot)
!status work-item 83-10-10
!references AI-00187, 83-00137, 83-00324, 83-00661, 83-00699, 83-00710
!topic Using a renamed package prefix inside a package

!summary 86-01-24

An expanded name is legal if the prefix denotes a package and the selector is a simple name declared within the visible part of the package, regardless of whether the prefix is a name declared by a renaming declaration.

!question 86-03-04

Consider the following example:

```
package P is
     X : INTEGER;
     package RENAMED_PACKAGE renames P;
     Y : INTEGER renames P.X;                  -- legal
     Z : INTEGER renames RENAMED_PACKAGE.X;    -- legal?
end P;
```

Rule (e) (paragraphs 4.1.3(14-15)) seems to allow the use of RENAMED_PACKAGE, but rule (f) (4.1.3(18)) disallows it. Which rule governs the interpretation of the above case?

!recommendation 86-01-24

A simple name declared in the visible part of a package specification can be the selector of an expanded name whose prefix is a name declared by a renaming declaration.

!discussion 86-01-01

In interpreting 4.1.3(14-15) and 4.1.3(18), one could take the position that the rules are to be understood as covering non-overlapping cases, i.e., rule (f) applies to situations not already covered by rule (e). With this viewpoint, the expanded name RENAMED_PACKAGE.X is legal. On the other hand, one could argue that the specific rule forbidding such an expanded name takes precedence over the more general rule given as rule (e).

A survey of 8 implementations showed that 7 allowed the use of a prefix declared by a renaming declaration as long as the selector was declared in

the visible part of the package denoted by the prefix, i.e., most of the existing implementations support the recommendation.

In short, since the Standard is unclear on the legality of such names, and since implementations tend to allow them, it is reasonable to interpret the Standard as allowing such names.

Ada Commentary ai-00018-ra.wj downloaded on Mon May 9 12:45:02 EDT 1988

!standard 04.03.02 (11) 86-07-23 AI-00018/06
!class ramification 86-02-20
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-20
!status work-item 83-10-10
!references 83-00130, 83-00131, 83-00132, 83-00003
!topic Checking aggregate index and subcomponent values

!summary 86-01-28

The check that the value of an index in an array aggregate belongs to an index subtype can be made before or after all choices have been evaluated. Similarly, the check that a subcomponent value belongs to the subcomponent's subtype can be performed before or after all subcomponent expressions have been evaluated.

!question 86-01-28

4.3.2(11) says:

> For the evaluation of an aggregate that is not a null array, a check is made that the index values defined by choices belong to the corresponding index subtypes, and also that the value of each subcomponent of the aggregate belongs to the subtype of this subcomponent. For an n-dimensional multidimensional aggregate, a check is made that all (n-1)-dimensional subaggregates have the same bounds.

In the evaluation of a multi-dimensional array aggregate:

1) must all choices be evaluated before making the checks specified in 4.3.2(11)?

2) must all component value expressions be evaluated before making the checks specified in 4.3.2(11)?

3) can all choices and expressions be evaluated before making any of the checks specified in 4.3.2(11)?

In other words, when do the subtype checks for the index values and subcomponent values have to be made with respect to the evaluation of a multi-dimensional array aggregate?

!response 86-03-05

When evaluating an aggregate, the Standard does not specify when the checks specified in 4.3.2(11) are made relative to the evaluations specified in 4.3.2(10). 4.3.2(10) simply specifies that no expression in a component

association can be evaluated until all choices have been evaluated; 4.3.2(11) simply specifies what checks must be made when values are known. In the absence of specific wording stating that the checks can only be made after all choices are evaluated, an implementation is free to make the checks as soon as possible.

The Standard does sometimes specify when checks are to be made relative to the evaluation of names or expressions (e.g., see 5.2(3)). The lack of such a specification in 4.3.2 should be considered intentional.

Ada Commentary ai-00019-bi.wj downloaded on Mon May 9 12:45:03 EDT 1988

!standard 04.03.02 (11) 86-07-23 AI-00019/07
!class binding interpretation 83-10-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-03-15
!status work-item 83-10-10
!references 83-00004, 83-00076, 83-00182, 83-00239, 83-00307
!topic Checking for too many components in positional aggregates

!summary 84-03-15

CONSTRAINT_ERROR is raised if the bounds of a positional aggregate do not belong to the corresponding index subtype.

!question 84-07-13

For non-null array aggregates, 4.3.2(11) requires that the index values defined by choices be checked to ensure they belong to the corresponding index subtypes. No check is specified for the bounds of a positional aggregate, leading to the following question:

```
type LITTLE is (x, y, z);
type UC is array (LITTLE range <>) of INTEGER;
... (1, 2, 3, 4) ...    -- is CONSTRAINT_ERROR raised?
```

If the aggregate, (1, 2, 3, 4), appears in a context where it is required to be of type UC, should CONSTRAINT_ERROR be raised because there is no valid upper bound for the aggregate?

!recommendation 84-03-21

For positional aggregates, a check is made that the index bounds belong to the corresponding index subtype; CONSTRAINT_ERROR is raised if this check fails.

!discussion 84-03-21

An array with too many components for the type UC could not be a value of type UC. (According to 3.3(1), "a type is characterized by a set of values and a set of operations" while 3.6(4) says that a one-dimensional array value "has a distinct component for each possible index value ... The possible values for a given index are all the values between the lower and upper bounds, inclusive.") There are many other instances in the language in which an expression of some type, if its evaluation were allowed to complete normally, would produce a value outside that type. In all these cases, the effect of evaluating such an expression is to raise an exception. For example, the literal 1E100, if used in an INTEGER context, involves an implicit conversion of the universal_integer 1E100 to INTEGER. At runtime,

when this conversion is attempted, the exception NUMERIC_ERROR would be raised if 1E100 > INTEGER'LAST.

Therefore, it is clearly intended that all aggregates producing values outside their type will raise an exception when evaluated. In particular, the example above will raise CONSTRAINT_ERROR upon evaluation because no values of type UC have four components. The effect is equivalent to requiring a check that the bounds of a positional array aggregate belong to the corresponding index subtype; CONSTRAINT_ERROR is raised when this check fails.

Ada Commentary ai-00020-ra.wj downloaded on Mon May 9 12:50:01 EDT 1988

!standard 04.05.05 (10) 86-07-23 AI-00020/07
!class ramification 84-02-06
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status work-item 83-10-10
!references 83-00128, 83-00129, 83-00201, 83-00207, 83-00208, 83-00217, 83-00248, 83-00603
!topic Real literals with fixed point multiplication and division

!summary 84-07-13

A real literal is not allowed as an operand of a fixed point multiplication or division. The possibility of adopting a more liberal rule in a future version of the language will be studied.

!question 84-07-13

Multiplication and division are allowed for operands of any fixed point type, but the Standard appears to forbid fixed point multiplications and divisions involving real literals. Is this an oversight?

!response 84-03-14

The expression

```
Fixed_Point_Type (V1 * 3.14)
```

is illegal when V1 has a fixed point type because * is not defined for one operand of type universal real and the other operand of some fixed point type; hence, the expression can only be legal if the real literal can be converted (implicitly) to a unique fixed point type [4.6(15)]. But there is no unique fixed point type to which 3.14 can be converted. (There is no unique fixed point type since there are always at least two fixed point types whose scope includes every compilation unit -- the type DURATION and the anonymous predefined fixed point type specified in 3.5.9(7)). The same argument applies to fixed point division.

It would be possible to extend the language so * and / were defined for combinations of fixed point types and universal real values. However, performing such arithmetic operations with the accuracy required by the Standard is not straightforward. A literal such as 3.14 must be treated as an exact quantity, i.e., as though its 'SMALL were 3.14. (Then 3.14 will be represented exactly as a model number.) Consequently, to evaluate V1 * 3.14 correctly requires the same support from a compiler as that required when the compiler has decided to support 'SMALL values that are not powers of 2. Since an implementation does not have to support the representation clause for 'SMALL (at least for values that are not powers of two; see 13.1(10)), it would be inconsistent to require such support in order to process V1 * 3.14.

Ada Commentary ai-00023-bi.wj downloaded on Mon May 9 12:50:03 EDT 1988

| !standard 04.09 (11) 86-07-23 AI-00023/06
!standard 03.05.04 (04)
!standard 03.05.07 (10)
!standard 03.05.09 (08)
!class binding interpretation 83-10-18
| !status approved by WG9/AJPO 86-07-22
| !status approved by Director, AJPO 86-07-22
| !status approved by WG9/Ada Board 86-07-22
| !status approved by Ada Board 86-07-22
| !status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status work-item 83-10-10
!references 83-00073, 83-00202, 83-00250, 83-00315
!topic Static numeric subtypes

!summary 84-03-14

Declarations containing integer and real type definitions declare static subtypes, e.g., given

```
type T is range 1..10;
```

T is a static subtype.

!question 84-07-13

The wording of 3.5.4(4) seems to indicate that given the declaration

```
type T is range 1..10;
```

T is not a static subtype, since conversions of static expressions are not static. Similar wording applies to the declaration of fixed and floating point types. Are all numeric types non-static?

!recommendation 83-10-30

It is the intent that an integer type declared with an integer type definition be considered static, and similarly, a real type declared with a real type definition should be considered static. The equivalence stated in 3.5.4(4), 3.5.7(10), and 3.5.9(8) should not be taken literally for purposes of defining whether the declared type is static.

!discussion 84-03-14

By 3.5.4(4), the declaration

```
type T is range 1..10;
```

is equivalent to

```
type integer_type is new predefined_integer_type;
subtype T is integer_type
    range integer_type(1) .. integer_type(10);
```

and, by 4.9, type conversions are not static. Hence, T is not a static subtype. Similar equivalences are defined for floating point and fixed point type declarations [3.5.7(10); 3.5.9(8)]. Interpreting these equivalences literally would imply that no user-defined integer or real subtype is static. 4.9(11), however, implies that real subtypes can be static by mentioning floating and fixed point constraints in association with the definition of scalar static subtypes. (Moreover, Ada comment #3077 shows that numeric types were intended to be static.)

The intent of the Standard is, therefore, that a subtype declared with an integer or real type definition be considered static, and that the equivalences stated in 3.5.4(4), 3.5.7(10) and 3.5.9(8) not be taken literally for purposes of defining whether the declared subtype is static.

The possibility of removing this restriction in a future version of the Standard will be studied, but the current Standard is unambiguous on this point.

Ada Commentary ai-00024-bi.wj downloaded on Mon May 9 13:45:02 EDT 1988

!standard 06.04.01 (04) 86-07-23 AI-00024/09
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status WG14/ADA Board approved (provisional) 84-11-27
!status WG14-approved (provisional) 84-11-27
!status committee-approved 84-06-28
!status work-item 83-10-10
!references 83-00069
!topic Type conversions as out parameters for non-scalar types

!summary 85-12-11

If an out parameter has the form of a type conversion and the type mark denotes an array type, the type conversion is performed before the call (see 4.6(11, 13) for the semantics of such a conversion). If the type mark denotes an access type, the value of the variable is converted before the call to the base type of the formal parameter; the designated object need not satisfy a constraint imposed by the formal parameter.

!question 86-05-20

If an actual out parameter has the form of a type conversion, 6.4.1(4) suggests, by omission, that the parameter is not converted before the call, since the Standard says, "... then before the call, for a parameter of mode IN OUT, the variable is converted to the specified type." However, other rules in the Standard suggest that such conversions must be performed for array and access types. Are conversions to array and access types performed before the call?

!recommendation 84-08-31

When an actual out parameter has the form of a type conversion and the type mark denotes an array type, the variable is converted to the specified type before the call (using the rules in 4.6(11, 13)). For parameters having an access type, the value of the actual parameter is converted to the base type of the formal parameter; no check is made to see whether the result of the conversion satisfies a constraint imposed by the type mark.

!discussion 85-12-11

The analysis of this problem is somewhat lengthy. Here is a summary:

. first we consider array types. Semantic inconsistencies arise when a formal out parameter is an array type unless the type conversion is performed before the call.

. then we consider access types. No inconsistencies arise if only the value is converted and certain checks normally performed for type conversions are omitted.

. for record types, it is impossible to tell whether the conversion is performed before the call or not;

. for scalar types, no semantic inconsistencies arise if the conversion is not performed before the call.

Our conclusion is that the Standard should be interpreted in a way that removes possible inconsistencies, and hence, the intent of the Standard is that actual out parameter type conversions are to be performed before the call when the formal parameter has an array type or access type, but that for access types, an object designated by the actual parameter need not satisfy a constraint imposed by the formal parameter.

ARRAY TYPES

If V is an array variable, and P is specified as:

```
P (F : out F_TYPE);
```

then a call of P using an actual parameter in the form of a type conversion:

```
P (F_TYPE(V));
```

is understood to update V.

So where 6.2(5) refers to "updating the value of the associated actual parameter", this must refer to V. And in the next two paragraphs, where it is explained how this updating may be achieved by copy or by reference, "the associated actual parameter" means V.

The updating involves the reverse conversion, from the subtype of the formal to that of V, but this conversion will usually be virtual (no code will be generated), since 6.2(9) tells us that (for unconstrained F_TYPE) "the bounds are obtained from the actual parameter, and the formal parameter is constrained by these bounds".

For this updating to be safe, all relevant checks must already have been performed on the validity of the conversion: checks on dimensionality, on index subtype convertibility and bounds as well as on component subtypes. This will also justify the omission of a check upon return from P (6.4.1(9)).

With mode in out there is no problem because the second sentence of 6.4.1(4) tells us that before the call "the variable is converted to the specified type" (which must be that of the formal parameter). The present difficulty arises because essentially the same conversion is still needed for mode out: although the value of V is not wanted, the validity of the conversion needs checking and (for unconstrained F_TYPE) the bounds must be obtained from V.

Consider first the example:

```
subtype SM_INT is INTEGER range 1 .. 3;
type A_LONG is array(LONG_INTEGER range <>) of INTEGER;
type A_INT is array(SM_INT range <>) of INTEGER;
procedure P_INT(X : out A_INT);

   -- define a constant one bigger than INTEGER'LAST
BIG : constant := INTEGER'POS(INTEGER'LAST) +1;

   -- declare an array with bounds too big for INTEGER representation
X_LONG : A_LONG(BIG .. BIG +2);
   ...
   -- call P_INT with a type conversion as actual parameter
P_INT(A_INT(X_LONG));
```

If the actual parameter type conversion of the variable X_LONG is not performed before the call, a semantic inconsistency arises, as discussed below.

The conversion has the following effects:

(1) For each index position the bounds of the result are obtained by converting the bounds of the operand to the corresponding index type of the target type (4.6(11)).

(2) For each index position a check is made that the bounds of the result belong to the corresponding index subtype of the target type (unless the operand is a null array) (4.6(13)).

(3) A check is made that any constraint on the component subtype is the same for the operand array type as for the target array type (4.6(13)). (Moreover it is a consequence of the 4.6(11) rule that the component types must be the same.)

(4) The value of each component of the result is that of the matching component of the operand. (Matching here is in the sense of 4.5.2(7), that is, the lower bounds must match.)

We have to decide which of these effects is needed in order to avoid semantic inconsistencies in the given situation.

For unconstrained formal array parameters, the Standard requires that the bounds of the formal parameter "be obtained from the actual parameter" (6.2(9)). But in the above case, the bounds of the variable X_LONG have a different base type than the bounds of the formal parameter, and moreover, the values of the actual parameter's bounds lie outside the range of the formal's index base type. There is no semantically consistent way the bounds for formal parameter X can be obtained without first converting the actual parameter bounds. (In this example, such a conversion will raise NUMERIC_ERROR, since BIG and BIG+2 lie outside the range of INTEGER, A_INT's index base type (3.5.4(10)).)

If X_LONG had the bounds 4..6, a different problem would arise since neither

4 nor 6 belong to A_INT's index subtype SM_INT. 6.2(9) requires, however, that the formal parameter be constrained by the bounds obtained from the actual parameter.

Conclusion: The effects (1) and (2) are needed to avoid the semantic inconsistency described in the example above: with the given bounds of X_LONG we get NUMERIC_ERROR by (1), and with the bounds 4..6 we get CONSTRAINT_ERROR by (2). Moreover, we should interpret paragraphs 6.2(5-7) as referring to the operand X_LONG of the type conversion when they mention the actual parameter. (It is the operand that gets updated as a result of the call.) The semantic inconsistency is also related to this.

Next consider the example:

```
type LONG_COMP  is array (INTEGER range <>) of INTEGER range -5 .. 5;
type SHORT_COMP is array (INTEGER range <>) of INTEGER range -2 .. 2;
SHORT : SHORT_COMP(1 .. 10);
procedure P_INT(X : out LONG_COMP);

   -- call P_INT with a type conversion
P_INT(LONG_COMP(SHORT));
```

Again, if the actual parameter type conversion is not performed before the call, an inconsistency would arise. Note first that in the case of array parameters without explicit type conversion no check is performed for the subtype of the formal and actual components. This does no harm when the actual and formal parameters have the same base type, since then the component subtypes are necessarily the same.

In the present case of a type conversion as actual parameter, however, the component subtypes could be different, as in the above example. Thus failure to check the component subtype constraints before the call could result in SHORT being assigned invalid values (whether the actual parameter is passed by reference or by copy).

In short, step (3) of the conversion must be performed before the call to avoid potential problems.

Step (4) of the conversion must be omitted; the use of a type conversion as an OUT actual parameter does not imply that a copy is made of the converted variable's value. After the required checks are performed, the conversion only determines how components of the actual parameter are matched with components of the formal parameter. (The fourth step is, of course, needed when the formal parameter's value is converted and assigned to the variable.)

Overall conclusion: If the array type conversion is done before the call, all potential inconsistencies disappear.

ACCESS TYPES

For access types as formal parameters, the Standard requires that the value of the actual parameter be copied to the formal parameter [6.2(6)]. If the

actual parameter has the form of a type conversion, the copying cannot be performed, in principle, when the base types of the formal parameter and actual variable are different: they have no values in common, so the required copy operation is meaningless. The copy operation is also potentially not value-preserving if the representations of the formal and actual access types are different (e.g., if one type has a shorter length than the other because it is an offset pointer). For example:

```
type AST is access STRING;
type N_AST is new AST(1..3);
X_AST : AST(3..5);
procedure PA (X : out N_AST);
...
PA (N_AST(X_AST));
```

The semantics of access type conversion require checking that the designated object satisfies any constraint specified for the target type [4.6(12) and 3.8(6)]. If this check is not performed for actual out parameters having an access type, no semantic inconsistencies result, since the object designated by the formal out parameter cannot be read [6.2(5)]. Moreover, when the subprogram returns, a check is made to ensure that the object designated by the formal parameter satisfies any constraint specified for the actual variable [6.4.1(7)]. Since failure to perform the check on the subtype of the designated object produces no semantic inconsistencies, it is possible, in principle, to say simply that the value of the actual variable is converted to the base type of the formal parameter and then copied to the formal parameter. No run-time checks need be performed.

RECORD TYPES

For record types, a representation conversion might be needed for out parameters, but such a conversion has no detectable semantic effects. Therefore, one is free to say the conversion is performed or not.

SCALAR TYPES

Since the value of a scalar out parameter is not copied to the formal parameter before the call, there is no need to evaluate the type conversion before the call.

SUMMARY

When a formal parameter has an array type and the actual parameter has the form of a type conversion, the type conversion must be performed (as defined in 4.6) before the call to avoid semantic inconsistencies. If the formal parameter has an access type, at least the access value must be converted to the formal's type before the call. For scalar types, no conversion need be done before the call. For record types, there is no detectable effect associated with type conversion -- it can be required before the call if this is convenient.

In short, to eliminate inconsistencies, the simplest interpretation of the

Standard's intent is that when an actual out parameter has the form of a type conversion, the type conversion is performed before the call for array types, and the result of the conversion is used as the actual parameter. For access types, only the value of the variable is converted; no constraint checks need be performed when the type conversion is performed.

Ada Commentary ai-00025-bi.wj downloaded on Mon May 9 13:45:05 EDT 1988

!standard 06.04.01 (09) 86-07-23 AI-00025/08
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status discussion corrected in accordance with AI-00396
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-02-06
!status work-item 83-10-10
!references AI-00396, 83-00072, 83-00316, 83-00510
!topic Checking out parameter constraints for private types

!summary 84-03-16

When a subprogram parameter has a private type, the constraint checks that are performed before or after the call are those appropriate for the type declared in the private type's full declaration.

!question 84-07-13

When an out parameter has a private type, the Standard appears to say that if the type has no discriminants, no constraint checks are made before or after the call. Should the checks required for private types be those appropriate for the full declaration?

!recommendation 84-01-19

When a parameter has a private type, the constraint checks that are performed before or after the call are those appropriate for the type declared in the private type's full declaration.

!discussion 86-07-23

The following example shows how a value can be assigned to an actual out parameter that violates the actual parameter's constraint if no check is performed after the return (as 6.4.1(9) suggests is the case for private types):

```
package P is
     type T is private;
     DC : constant T;
     generic package PP is
     end PP;
private
     type T is new INTEGER;
     DC : constant T := -1;
end P;
```

```
     procedure Q (X : out P.T) is
     begin
          X := P.DC;
     end Q;

     generic
          Y : in out P.T;
     package CALL is
     end CALL;

     package body CALL is
     begin
          Q (Y);
     end CALL;

-- note Q has a variable of a private type as an out parameter.  If we
-- interpret 6.4.1(9) literally, the value of Y is checked before the
-- call and not after the return.

     package body P is
          Z : T range 0..1 := 0;
          package body PP is
               package CALL_Q is new CALL(Z);
          end PP;
     end P;

     package CALL_Q_NOW is new P.PP;
```

The effect of elaborating CALL_Q_NOW is to assign an invalid value to P.Z if no check is made upon the subprogram's return.

6.2(8) says, "For a parameter whose type is a private type, the above effects are achieved according to the rule that applies to the corresponding full type declaration." "Above effects" refers to whether a parameter is passed by copy or not, and does not clearly include the constraint checking "effects" of 6.4.1. Clearly, however, it would be undesirable (and inconsistent with the intent of the language design) not to check the value of the formal parameter after the call returns, in the above case. So the Standard must be interpreted to say that parameters of private types are checked in the manner that is appropriate for the full declaration of the type. For purposes of parameter passing semantics, 6.2(8) should be understood to imply there is no such thing as a parameter of a private type, so 6.4.1(9) refers to array, record, and task types only.

Ada Commentary ai-00026-co.wj downloaded on Mon May 9 13:50:02 EDT 1988

!standard 07.04.02 (09) 87-06-18 AI-00026/07
!class confirmation 86-08-07
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
!status work-item 86-07-31
!status received 84-01-29
!references 83-00115, 83-00116
!topic Effect of full type decl on CONSTRAINED attribute

!summary 86-09-12

After the full declaration of a private type P, P'CONSTRAINED is not allowed unless P's full declaration derives from a private type.

!question 86-08-12

Consider the following example:

```
package A is
    type P is private;                  -- declares P'CONSTRAINED
private
    type P is new INTEGER;              -- P is no longer private
    ...
    C1 : BOOLEAN := P'CONSTRAINED; -- illegal? (yes)
```

After the full declaration, P is no longer a private type, but since the CONSTRAINED operation is visible, can it nonetheless be applied to P after the full declaration?

!response 86-09-15

Visibility is a necessary but not a sufficient condition for legal use of an operation. 7.4.2(9-10) requires that if the prefix of the attribute CONSTRAINED denotes a type, the type must be a private type. In the example given in the question, the prefix P no longer denotes a private type after P's full declaration, and so P'CONSTRAINED is not allowed, even though the attribute itself is visible. (A full declaration of a private type can only declare a private type by deriving from a private type.)

This treatment of the CONSTRAINED attribute is consistent with the treatment of other operations. For example, having an assignment operation visible does not mean it can be applied to a constant. Similarly, even if the "+" operator for INTEGERs is visible, it cannot be used if one of the operands is a formal parameter of mode OUT.

Ada Commentary ai-00027-bi.wj downloaded on Mon May 9 13:50:04 EDT 1988

!standard 08.03 (18) 87-06-18 AI-00027/07
!class binding interpretation 86-09-11
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (22-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
!status work-item 83-10-10
!references 83-00136, 83-00136, 83-00696
!topic Visibility of type mark in explicit conversion or qualified expression

!summary 86-09-13

The type mark that occurs in an explicit conversion or in a qualified expression must denote a visible declaration.

!question 86-09-13

3.3.3(6) says that explicit conversion and qualification are basic operations.

8.3(18) says that the notation for a basic operation is directly visible throughout its scope:

> Finally, the notation associated with a basic operation is directly visible within the entire scope of this operation.

Consider the following example:

```
declare
    package PAC is
        type INT is range 1..10;
    end PAC;

    X : INTEGER := 1;
    Y : PAC.INT;
begin
    Y := INT(X);          -- legal? (no)
    Y := INT'(Y);         -- legal? (no)
    Y := PAC.INT(X);      -- legal
end;
```

If the notation for explicit conversion is directly visible throughout its scope, then why shouldn't INT(X) be legal as well as PAC.INT(X), and similarly, INT'(Y).

!recommendation 87-02-23

The type mark that occurs in an explicit conversion or in a qualified expression must denote a visible declaration.

!discussion 87-02-23

The rule in 8.3(18) specifies when basic operations can be used. The rule applies to operations such as assignment and component selection as well as to conversion and qualification. The rule is not intended to affect the normal visibility rules for identifiers. Instead, it is intended to supplement the other rules by defining the meaning of syntactic constructs associated with basic operations. The conversion and the qualified expression given in the question are illegal because the identifier INT must be associated with a declaration by the other visibility rules. Since no declaration of INT is directly visible, the conversion and the qualified expression are intended to be considered illegal.

Although no declaration of INT is directly visible, the conversion and qualification operations are visible and can be used. For example:

```
subtype PINT is PAC.INT;
Z1 : PAC.INT := 5;          -- implicit conversion
Z2 : PAC.INT := PINT(5);    -- explicit conversion
Z3 : PAC.INT := PINT'(5); -- qualification
```

The explicit conversion and the qualified expression using PINT are legal because PINT is directly visible and because the conversion (as well as the qualification) operation associated with PINT's base type is visible (by 8.3(18)).

Ada Commentary ai-00030-ra.wj downloaded on Mon May 9 14:45:02 EDT 1988

!standard 09.07.01 (05) 86-07-23 AI-00030/07
!class ramification 86-02-20
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-20
!status work-item 83-10-10
!references 83-00125, 83-00126, 83-00127, 83-00318
!topic All guards need not be evaluated first

!summary 86-03-05

In the execution of a selective wait statement, the evaluation of conditions, delay expressions, and entry indices is performed in some order that is not defined by the language, except that a delay expression or an entry index cannot be evaluated until after the condition for the corresponding alternative is evaluated and found to be true.

!question 86-01-16

When evaluating the expressions in a selective wait, must the delay expressions and entry family indexes be evaluated immediately after determining that an alternative is open, or are all the conditions evaluated first, or is the order undefined?

!response 86-03-05

9.7.1(5) says:

> For the execution of a selective wait, any conditions specified after WHEN are evaluated in some order that is not defined by the language; open alternatives are thus determined. For an open delay alternative, the delay expression is also evaluated. Similarly, for an open accept alternative for an entry of a family, the entry index is also evaluated. Selection and execution of one open alternative, or of the else part, then completes the execution of the selective wait; ....

By saying the delay expression is "also" evaluated, one might argue that this means immediately after determining whether the associated alternative is open. On the other hand, one might argue that since the standard says "any conditions" (note use of the plural), and since only a single condition can follow a single WHEN, the implication is that all conditions are evaluated to determine which alternatives are open, and then, delay expressions and entry family indexes are evaluated for open alternatives.

Possible interpretations are:

1. all conditions are evaluated first; or

2. evaluation of a condition must be followed by evaluation of a delay expression or index; or

3. evaluation of conditions, delays, and indexes can proceed in any order as long as no delay expression or index is evaluated for a closed alternative.

When a specific order is intended, the standard always uses phrases like "... first evaluated ... then ... finally ...". Thus, the current wording supports interpretation 3, which is the least restrictive.

Ada Commentary ai-00031-co.wj downloaded on Mon May 9 14:45:05 EDT 1988

!standard 09.08 (01) 86-07-23 AI-00031/06
!class confirmation 84-02-06
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status work-item 83-10-10
!references 83-00021, 83-00203
!topic Out-of-range argument to pragma PRIORITY

!summary 84-03-16

If the argument to pragma PRIORITY does not lie in the range of the subtype PRIORITY, the pragma has no effect.

!question 84-07-13

Suppose that subtype SYSTEM.PRIORITY is INTEGER range 1..10. Then if a unit contains

```
pragma PRIORITY (11);
```

which of the following are possible actions?

(1) The pragma is ignored.
(2) CONSTRAINT_ERROR or some other exception is raised when the unit is elaborated.
(3) The program is illegal.
(4) The program is erroneous and PROGRAM_ERROR may be raised.

!response 84-03-17

2.8(9) states that a pragma has no effect if "its arguments do not correspond to what is allowed for the pragma." 9.8(1) states that a task's priority "is a value of subtype PRIORITY ..." It is clearly the intent of the RM to limit the allowable values to those belonging to the subtype PRIORITY. Hence, the pragma has no effect if the value is outside the range of the subtype PRIORITY.

Ada Commentary ai-00032-ra.wj downloaded on Mon May 9 14:50:02 EDT 1988

!standard 09.08 (04) 87-03-16 AI-00032/09
!class ramification 85-05-18
!status approved by WG9/AJPO 87-03-10
!status approved by Director, AJPO 87-03-10
!status approved by Ada Board (14-4-3) 87-02-19
!status approved by WG9 85-11-18
!status committee-approved (reviewed) 85-09-04
!status committee-approved (reviewed) 85-05-18
!status committee-approved (7-0-2) 85-02-27
!status work-item 85-01-31
!status received 84-01-29
!references AI-00045, 83-00059, 83-00517, 83-00518
!topic Preemptive scheduling is required

!summary 87-02-23

If an implementation supports more than one priority level, or interrupts, then it must also support a preemptive scheduling policy.

!question 85-01-31

If a task is executing when a task with a higher priority becomes executable because a delay has expired, must execution of the lower priority task cease immediately so execution of the higher priority task can continue?

!response 87-02-23

9.8(4) says:

> If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same physical processors and the same other processing resources, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not.

Expiration of a delay means a task is eligible for execution. If an implementation determines that a delay has expired for a high priority task, then that task must be scheduled for execution. If the physical processor and other processing resources needed to execute the higher priority task are currently being used by a task of lower priority, execution of the lower priority task must be suspended.

If preemptive scheduling is not feasible in a particular implementation, then the implementation should not provide multiple priority levels (see AI-00045) or support interrupts.

(Note: the phrase "could sensibly be executed" refers to situations in which the high priority task can actually make use of the processor and other resources being used by the lower priority task. In some distributed processing situations, a high priority task may not be able to execute on some processors. Preemption is only required for processing resources the high priority task can use.)

Ada Commentary ai-00034-ra.wj downloaded on Mon May 9 14:50:03 EDT 1988

!standard 09.09 (06) 86-07-23 AI-00034/06
!class ramification 84-10-30
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-03-26
!status received 84-01-29
!references 83-00035
!topic Value of COUNT in an accept statement

!summary 84-12-10

In an accept statement for a member of an entry family, the member being called (and consequently, the task calling the member) is not known until the entry index has been evaluated. This means that if the entry index contains a COUNT attribute, its value is not affected by what member of the family is eventually determined to be called.

!question 84-12-10

Evaluation of an accept statment for an entry family member means evaluating the entry family index [9.5(10)]. 9.9(6) says "if the [COUNT] attribute is evaluated by the execution of an accept statement for the entry [named in its prefix], the count does not include the calling task." But consider an entry family E indexed by some integer type. Suppose there is one task waiting on E(1) and one waiting on E(0). Which task is chosen by

```
accept E (E(1)'COUNT);
```

E(1)'COUNT evaluates to 1, so E(1) is chosen; but in that case, the calling task is not included: so E(1)'COUNT must evaluate to 0, thus choosing E(0). But if we choose E(0), the calling task to E(1) must be included in the count, so E(1)'COUNT must evaluate to 1... Hence, E(1)'COUNT does not have a consistent value unless the calling task is included when the COUNT attribute is evaluated. If the COUNT attribute is used in an entry family index of an accept statement, shouldn't the calling task be included in the returned value?

!response 84-12-10

9.9(6) says the value of the COUNT attribute does not include the calling task "if the attribute is evaluated by the execution of an accept statement for the entry" named in the prefix. When the accept statement is for an entry family, evaluation of the entry name includes evaluation of the entry index. Until the index has been evaluated, it is not known which member of the family is being named. Consequently, no calling task can be excluded from the value of the COUNT attribute. In the above example, E(1)'COUNT evaluates to one, so entry family member E(1) is called. Subsequent execution of the accept statement reflects the fact that the calling task has been removed from the queue for this entry family member.

In short, while evaluating an entry family index, there is no calling task that can be excluded from the count. Of course, once a call has been accepted, the calling task is removed from the queue, and so is no longer included in the value of the attribute.

Ada Commentary ai-00035-ra.wj downloaded on Mon May 9 15:45:02 EDT 1988

!standard 10.02 (03) 86-12-04 AI-00035/06
!class ramification 84-03-26
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved (5-0-0) 86-09-11
!status work-item 84-03-26
!status received 84-01-29
!references 83-00067, 83-00068, 83-00109, 83-00259, 83-00314, 83-00334
!topic Body stubs are not allowed in package specifications

!summary 84-03-26

Body stubs are not allowed in package specifications.

!question 86-12-04

10.2(3) says:

> A body stub is only allowed as the body of a program unit ... if the body stub occurs immediately within either the specification of a library package or the declarative part of another compilation unit.

This apparently contradicts 7.1(5):

> A body is not a basic declarative item and so cannot appear in a package specification.

!response 84-03-26

There is no contradiction here. The condition stated in 10.2 (3) is satisfied only if the second alternative holds since the first alternative is always false. Hence, that alternative in no way affects the meaning of 10.2(3). The phrase, "either the specification of a library package or", can be ignored. It was accidentally left in the Standard when an earlier wording of the rule was revised.

Ada Commentary ai-00037-bi.wj downloaded on Mon May 9 15:45:04 EDT 1988

!standard 12.03.02 (04) 86-12-01 AI-00037/12
!class binding interpretation 85-02-25
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status approved by WG9 86-05-09 (provisional)
!status WG9/ADA Board approved (provisional) 85-05-13
!status committee-approved (10-1-1) 85-05 (by letter ballot)
!status committee-approved (8-1-0) 85-02-25
!status work-item 84-03-26
!status received 84-01-29
!references 83-00121, 83-00122, 83-00123, 83-00124, 83-00398, 83-00415, 83-00419
!topic Instantiating when discriminants have defaults

!summary 86-09-18

An actual subtype in a generic instantiation can be an unconstrained type with discriminants that have defaults even if an occurrence of the formal type (as an unconstrained subtype indication) is at a place where either a constraint or default discriminants would be required for a type with discriminants.

!question 85-11-05

Paragraph 12.3.2(4) appears to forbid certain generic instantiations with types that have discriminants, whether or not the discriminants have defaults. This restriction seriously limits the utility of generic units, and is surely a mistake. The importance of the limitation can be seen in the following example. Consider a generic unit that is intended to implement a FIFO queue for a record type. Such a generic unit might implement the queue as an array:

```
generic
     type ELEMENT_TYPE is private;
     QUEUE_SIZE : POSITIVE;
package FIFO_QUEUE is
     type FIFO_QUEUE_TYPE is private;
     -- operations on FIFO_QUEUE_TYPE
private
     type CONTENTS_TYPE is array (1..QUEUE_SIZE) of ELEMENT_TYPE;
     type FIFO_QUEUE_TYPE is
          record
               CONTENTS : CONTENTS_TYPE;
               SIZE     : INTEGER range 0..QUEUE_SIZE := 0;
          end record;
end FIFO_QUEUE;
```

Now suppose we wish to have a queue of varying length strings, and the varying length string type has been declared as:

```
subtype MAX_STRING is INTEGER range 0 .. 10_000;
```

```
type TEXT (LENGTH : MAX_STRING := 0) is
     record
          VALUE : STRING (1..LENGTH);
     end record;

package FIFO_TEXT_QUEUE is new FIFO_QUEUE (TEXT, 100);    -- illegal
```

12.3.2(4) says that for an instantiation, "The actual subtype [i.e., TEXT in this case] must not be an ... unconstrained type with discriminants if any of [the formal parameter's occurrences as an unconstrained subtype indication] is at a place where either a constraint or default discriminants WOULD BE REQUIRED [emphasis added] ... for a type with discriminants." The use of ELEMENT_TYPE to declare an array component type is a place where either a constraint or default discriminants are required if the component type has discriminants [3.7.2(8)], so the instantiation seems to be illegal. If the queue is implemented as a list, the same problem arises:

```
private
     type FIFO_ELEMENT_TYPE is
          record
               VALUE : ELEMENT_TYPE;
               NEXT  : FIFO_QUEUE_TYPE;
          end record;
     type FIFO_QUEUE_TYPE is access FIFO_ELEMENT_TYPE;
end FIFO_QUEUE;
```

An instantiation with an unconstrained type is still illegal since the record component, VALUE, requires either a constrained type or a type with default discriminants.

Note that a non-generic package can be used to create a FIFO queue for types with discriminants if the discriminants have defaults. It's just that such a queue cannot be created by instantiating a generic unit.

Was it really intended that such instantiations be forbidden? Would it be possible to relax the rule?

!recommendation 86-05-20

For occurrences of the name of a formal private type at places where this name is used as an unconstrained subtype indication, the actual subtype can be an unconstrained type with discriminants that have defaults even if an occurrence of the formal type is at a place where either a constraint or default discriminants would be required for a type with discriminants. The same applies to occurrences of the name of a subtype of the formal type, and to occurrences of the name of any type or subtype derived, directly or indirectly, from the formal type.

!discussion 85-12-02

This discussion is in three parts: first we discuss alternative approaches for coping with the restriction stated in 12.3.2(4); second, we discuss the

reasons for the restriction; and finally, we discuss why it is felt that the rule should be relaxed.

LIVING WITH THE RULE

In general, to allow instantiations with unconstrained types, an implementation must use an access type. For example, the full declaration of the FIFO_QUEUE_TYPE for an array implementation would have to be:

```
private
     type ACC_ELEMENT_TYPE is access ELEMENT_TYPE;
     type CONTENTS_TYPE is array (1..QUEUE_SIZE) of ACC_ELEMENT_TYPE;
     type FIFO_QUEUE_TYPE is
          record
               CONTENTS : CONTENTS_TYPE;
               SIZE     : INTEGER range 0..QUEUE_SIZE := 0;
          end record;
end FIFO_QUEUE;
```

The operation for inserting an element in the QUEUE will be implemented as:

```
procedure INSERT (QUEUE : in out FIFO_QUEUE_TYPE;
                  VALUE : ELEMENT_TYPE) is
begin
     ...
     QUEUE.CONTENTS(QUEUE.SIZE) := new ELEMENT_TYPE'(VALUE);
```

This approach allows an instantiation with any unconstrained type, including an unconstrained array type, and is the approach that must be used to allow instantiations with the broadest possible range of non-limited types. Note that the declaration of ACC_ELEMENT_TYPE is legal for any instantiation since the type designated by an access type need not be constrained. Similarly, the type mark in an allocator need not denote a constrained type.

If you do not wish to use an access type implementation for every private type declared in a generic package, then you will need to support two implementations of each template -- one that uses access types (and which can therefore be instantiated for any unconstrained type), and one that does not. This doubling of code to be written and maintained can be a considerable implementation and maintenance burden.

RELAXING THE RULE

Any unconstrained array type or unconstrained type with discriminants that do not have defaults can be encapsulated as a record type that has default discriminants, e.g.:

```
subtype MAX_STRING is INTEGER range 0 .. 10_000;
type STR (L, R : MAX_STRING := 0) is
     record
          C : STRING (L..R);
     end record;
```

If the presence of default discriminants would make an instantiation legal, one could use a single generic template and still support instantiations with essentially arbitrary types. An organization that has implemented a large amount of Ada code has reported that this approach was successful until they started to use a compiler that correctly implemented the rule in 12.3.2(4).

MOTIVATION FOR THE RULE

The language problem that was addressed by the restriction in 12.3.2(4) is illustrated by the following example:

```
generic
     type T is private;
package P is
     type PRIV is private;
private
     type PRIV is new T;
end P;
```

Now consider the following declarations:

```
type T_Def (D : INTEGER := 0) is ... ;
type T_No_Def (D : INTEGER) is ... ;
```

and the instantiations:

```
package P_Def is new P (T_Def);
package P_No_Def is new P (T_No_Def);
```

If both instantiations are legal, is this declaration to be considered legal?

```
X : P_No_Def.PRIV;       -- legal?
```

Note that for the P_No_Def instantiation, the full declaration of PRIV declares it to be a type whose discriminants have no defaults. If the declaration of X is legal, the discriminants for X have no values, and this violates a fundamental design goal, namely, that discriminants always have defined values. There is no difficulty with the declaration:

```
Y : P_Def.PRIV;
```

since Y's discriminants have default values.

A derived type declaration as the full declaration for a private type is forbidden if the parent type has discriminants [7.4.1(3)], e.g., the full declaration:

```
type PRIV is new T_Def;     -- illegal
```

would be illegal. It was argued in comment #3659 that there is really no problem in allowing the P_Def instantiation, but if P_Def were allowed, then it would be consistent to also allow such derivations for private types (see comment #2696 for 7.4.1(3)).

Late in the language design phase, it was decided that the full declaration of a private type could not be an unconstrained type with discriminants, whether or not the discriminants had defaults. For consistency, it was decided that this rule should also be enforced for generic instantiations, so the rule in 12.3.2(4) was not conditioned on the presence or absence of defaults for discriminants. The issues raised by the question were not explicitly considered. In particular, it was not considered that an instantiation of a template could be considered illegal (if an actual parameter were a type with discriminants that have defaults) even though the equivalent package or subprogram declaration would be legal. Forcing the use of an access type implementation just to support instantiations with types that have discriminants with defaults was also not given any consideration.

CONCLUSION

It appears the restriction imposed by 12.3.2(4) is a serious error affecting the usability of generic units. Relaxing the restriction will not invalidate any existing legal Ada programs. Moreover, it should not be difficult to change implementations to allow instantiations when an actual unconstrained type has default discriminant values. Supporting the use of generic units is so important that this interpretation should be considered binding on implementors.

Given the current schedule of validation test releases, this binding interpretation will be enforced by validation tests beginning June 10, 1986.

Ada Commentary ai-00038-ra.wj downloaded on Mon May 9 15:50:01 EDT 1988

!standard 12.03.06 (02) 87-02-23 AI-00038/06
!class ramification 83-10-10
!status approved by WG9/AJPO 87-02-20
!status approved by Director, AJPO 87-02-20
!status approved by Ada Board (21-0-0) 87-02-19
!status approved by WG9 86-05-09
!status committee-approved (8-0-1) 85-11-20
!status work-item 83-10-10
!references 83-00087, 83-00088, 83-00673
!topic Declarations associated with default names

!summary 84-04-13

The normal visibility rules apply to identifiers used in default subprogram names, i.e., these identifiers are associated with declarations visible at the point of the generic declaration, not those visible at each place of instantiation.

!question 85-12-16

In what context are the default subprogram/entry names to be resolved relative to generic formal subprograms -- at the place of the generic declaration or at the place of each instantiation?

12.3.6(2) says "evaluation of [a default subprogram name] takes place during the elaboration of each instantiation that uses the default". Does this mean that binding of a default name to a declaration occurs at the place of each instantiation? Note also that 12.3(17) says that "for each omitted generic association (if any), the corresponding ... default name is evaluated."

!response 86-05-20

Evaluation of a name is described in RM 4.1(9). The time at which a default subprogram name is evaluated only matters when the name denotes an entry of a task or a member of an entry family. For example, if A is an array of tasks, the default name A(I).E is evaluated at the point of instantiation, i.e., the value of the subscript is determined.

Before evaluation (at run-time) can occur, however, the visibility rules determine (at compile-time) the declaration(s) denoted by an identifier. The visibility rules for default formal subprogram names are the same as for any other name in a generic unit, namely, a matching declaration of the name must be visible at the place where the name is used in the generic declaration. If more than one name is visible, the usual overloading resolution rules are applied (at compile-time). Visibility of names at the place of an instantiation is irrelevant, except when the default subprogram parameter has the form <>, in which case a subprogram or entry directly visible at the place of the instantiation is used (12.3.6(3)).

To summarize, the declaration denoted by a default subprogram name is determined in the context of the generic declaration (at compile-time); such a name is evaluated (at run-time), when needed, at the point of the instantiation.

Ada Commentary ai-00039-bi.wj downloaded on Mon May 9 15:50:02 EDT 1988

!standard 13.01 (06) 86-07-23 AI-00039/12
!standard 13.01 (07)
!standard 07.04.01 (04)
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (9-0-0) 85-05-18
!status work-item 84-11-05
!status received 84-01-29
!references AI-00322, 83-00256, 83-00361, 83-00390, 83-00031, 83-00451, 83-00483, 83-00507, 83-00533, 83-00657
!topic Forcing occurrences and premature uses of a type

!summary 85-09-19

Each operand of a relational operator (and similarly, the operand of a type conversion or membership test) is considered to be implicitly qualified with the name of the corresponding operand type; such implicit occurrences are considered to be occurrences of the type name with respect to the rules given in 13.1(6), 13.1(7), and 7.4.1(4). This means that such occurrences can be illegal if the implicit type name is an incompletely declared private type (7.4.1(4)), or they can make the subsequent occurrence of a representation clause illegal (13.1(6, 7)).

!question 85-07-26

3.8.1(4), 7.4.1(4), and 13.1(6) attempt to prevent uses of an entity before its representation is fully determined by a representation clause or by the full declaration of an incompletely declared type. The current rules do not fully cover the intended set of situations. In particular, it is possible to construct examples using a relational operator, a type conversion, or a user-defined operation that may have been intended to be illegal.

!recommendation 85-09-19

Each operand of a relational operator (and similarly, the operand of a type conversion or membership test) is considered to be implicitly qualified with the name of the corresponding operand type; such implicit occurrences are considered to be occurrences of the type name with respect to the rules given in 13.1(6), 13.1(7), and 7.4.1(4).

!discussion 85-10-02

The problems raised (see later examples for details) have much in common with what was called the "Brosgol Anomaly" or "Brosgol Problem" back in 1981. For the historically minded, NOTE-002, NOTE-121, LSN-166, LSN-183, LSN-184 and LSN-273 are the most informative as to the statement of the problem and its solution. Briefly, the problem is that in the linear elaboration model some uses of an entity require properties of the entity that may be later

specified (changed) by a representation clause for that entity. Nearly all problems of this kind arise for types, because most representation clauses apply to types. To deal with these problems, the Standard gives the following rules:

RM 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 to the use of a name that denotes the private type or a subtype of the private type and, likewise, to the use of a name that denotes any type or subtype that has a subcomponent of the private type. The only allowed occurrences of such a name are in a deferred constant declaration, a type or subtype declaration, a subprogram specification, or an entry declaration; moreover, occurrences within derived type definitions or within simple expressions are not allowed."

RM 13.1(6,7):

> "In the case of a type, certain occurrences of its name imply that the representation of the type must already have been determined. Consequently these occurrences force the default determination of the representation of any aspect of the representation not already determined by a prior type representation clause. This default determination is also forced by similar occurrences of the name of a subtype of the type, or of the name of any type or subtype that has subcomponents of the type. A forcing occurrence is any occurrence 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.
>
> "A representation clause for a given entity must not occur after an occurrence of the name of entity if this occurrence forces a default determination of representation for the entity."

The historical NOTEs and LSNs discussed the problem. LSN-166 formulated a solution (restrictions) based on the principle that an operation may not be invoked prior to its complete elaboration. This principle was then reduced to rules involving checks of the text of a program, since elaboration occurs at run time. The rules did not cover all the intended cases, as illustrated by some of the following examples.

Example 1 -- use of a relational operator:

```
procedure PROC is

     type T is delta 1.0 range -10.0 .. 10.0;

     OBJ : constant BOOLEAN := PROC."="(1.1, 1.0);
```

```
        for T'SMALL use 10.0**(-BOOLEAN'pos(OBJ));--'SMALL is 1.0 or 0.1

    begin...
```

Note that the real literals are implicitly converted to type T and PROC."=" is the predefined equality operator for T. Thus, the initial value expression of OBJ is static. The value of OBJ determines the model numbers for T, but the value of OBJ may be affected by what the model numbers are, i.e., if T'SMALL is 0.1 then OBJ must be FALSE, which means T'SMALL must be 1.0. If T'SMALL is 1.0, then OBJ is allowed to have the value TRUE, which implies T'SMALL has the value 0.1. The only consistent set of values is for OBJ to be FALSE and T'SMALL to equal 1.0.

Example 2 -- use of a basic operation, aggregate formation:

```
    package NASTY is

        type T is private;

        package INNER is
            type R (B : BOOLEAN) is
                record
                    case B is
                        when FALSE => R1 : INTEGER;
                        when TRUE  => R2 : T;
                    end case;
                end record;
        end INNER;

        B : BOOLEAN := INNER."="((FALSE, 1), (FALSE, 1));       -- Legal ?

    private

        type T is array (FALSE..B) of INTEGER;

    end;
```

Note that in the declaration of B, neither the name of type R nor of type T actually occurs, but overloading resolution determines the type of the aggregate to be T. The aggregate operation is called and expected to produce a value even though one of the component types has not yet been fully declared. Is an implementation really required to support operations on types whose representation is only partially determined? (Note that if the aggregate were explicitly qualified, the use of the type mark in the qualification would be a forcing occurrence.)

Example 3 -- use of a user-defined operation:

```
    type T is ...
    function F return T;
    function G (X : T) return BOOLEAN;
```

```
type R is
     record
          Z : BOOLEAN := G(F);      -- forcing occurrence?
     end record;
```

Note that PROGRAM_ERROR will be raised if R is used to declare an object before G's body has been elaborated.

Example 4 -- size of an incomplete type:

```
package PACK is

     type PRI is private;

private

     type SEP;
     type PRI is access SEP;

     X : PRI;                  -- initialized to null

     I : INTEGER := X.all'SIZE;                -- Legal?
end;
```

Is the initialization of variable I legal? One would hope that it is not because it would seem to violate the purpose of the rules of 13.1(6) regarding uses of a type before its representation has been determined and/or 3.8.1(4) regarding uses of an incomplete type before its full type occurs. However, both of those sections are formulated in terms of names that denote the type -- and this example nowhere names the type SEP in the initialization of I.

One might argue that the question is really moot because X.all must necessarily raise CONSTRAINT_ERROR (there is no way to initialize or otherwise set X with a non-null value prior to the full type of SEP in the absence of suppressing PROGRAM_ERROR for a function call.) But that merely says that, IF LEGAL, the value of attribute SIZE need not be defined by the language. However, the question of legality remains.

A careful analysis of all the situations in which these sorts of problems arise shows that the current rules fail to achieve the intent only in the case of operands of relational operators, type conversions, and membership tests (see comment 83-00533). Examples 1 and 2 are considered illegal according to the proposed rule because of the use of the relational operator "=".

Example 3 is legal. It is harmless because the default expression is not evaluated when R's declaration is elaborated, and PROGRAM_ERROR will be raised if an object declaration (without an explicit initialization) is elaborated before F or G's body is elaborated.

Example 4 is legal and will raise CONSTRAINT_ERROR when X.all is evaluated

since X has the value null. (4.1(10) says that CONSTRAINT_ERROR is not raised for the prefix of a representation attribute if "the value of the prefix is a null access value," but X.all has no value at all, so the rule does not apply.)

Ada Commentary ai-00040-ra.wj downloaded on Mon May 9 16:45:03 EDT 1988

!standard 13.01 (03) 86-12-01 AI-00040/07
!standard 13.06 (01)
!class ramification 83-10-10
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (8-0-1) 86-05-12 (pending editorial review)
!status work-item 84-04-13
!status received 84-01-29
!references AI-00138, 83-00032, 83-00204, 83-00325
!topic Multiple specification of T'SIZE, T'STORAGE_SIZE, T'SMALL

!summary 86-04-14

For a given type, the 'SIZE, 'STORAGE_SIZE, and 'SMALL attributes can each be specified at most once by an explicit representation clause.

!question 83-10-10

There seems to be no rule that forbids the following.

```
for T'SIZE use 123;
for T'SIZE use 456;
```

In fact, 13.1(3) explicitly allows more than one length clause for any given type. It is clear what the meaning is when, for example, both T'SIZE and T'SMALL are specified; it is not clear what the meaning of the above example is.

Are any of the following interpretations permitted? Are any of them required?

(1) It is not legal to specify the same attribute of the same type in more than one length clause.

(2) It is legal, but the specified values must be the same.

(3) It is legal, and the value actually used depends on the implementation.

(4) It is legal, and some particular one of the values must be used by all implementations. Note that in the case of T'STORAGE_SIZE, the values need not be static and so the length clause whose value actually gets used can't be chosen at compile time.

!response 86-04-14

13.6(1) states that "at most one representation clause is allowed for a given type and a given aspect of its representation." 'SIZE, 'STORAGE_SIZE, and 'SMALL specify different aspects of a type's representation. Each of these aspects can be specified at most once by an explicit representation clause,

in accordance with 13.6(1). 13.1(3)'s statement that "more than one length clause can be provided for a given type" must be understood in conjunction with 13.6(1), e.g., T'SIZE and T'SMALL can both be specified for a fixed point type, but T'SIZE cannot be specified twice for the same type, even if the same value is given both times. (Note: an explicit clause can be given even if an implicit clause is present; see AI-00138).

Ada Commentary ai-00045-co.wj downloaded on Mon May 9 16:45:04 EDT 1988

!standard 13.07 (02) 86-07-23 AI-00045/05
!class confirmation 84-02-06
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status work-item 83-10-10
!references 83-00027
!topic Subtype SYSTEM.PRIORITY

!summary 84-03-16

The subtype SYSTEM.PRIORITY can be non-static. The subtype can have a null range.

!question 84-07-13

May the range of subtype PRIORITY be null?

Must subtype PRIORITY be static?

!response 84-01-29

The Standard places no limitations on the definition of subtype PRIORITY. Hence, it may be null or non-static. Note that if the range is non-static, the values belonging to the subtype PRIORITY cannot be determined, in general, until run-time. This means it cannot be determined until run-time whether an allowable value has been given in the pragma, i.e., it cannot be determined until run-time whether the pragma should be ignored (sec 2.8(9)).

Making the subtype PRIORITY null means all pragma PRIORITYs will be ignored. This effect is not much different from the effect of declaring subtype PRIORITY to have a single value, which is clearly not forbidden by the Standard.

Ada Commentary ai-00046-ra.wj downloaded on Mon May 9 16:50:01 EDT 1988

!standard 14.02.01 (03) 86-07-23 AI-00046/06
!standard 14.02.01 (22)
!class ramification 84-02-06
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-06-29
!status board returned to committee 84-06-29
!status committee-approved 84-02-06
!status received 84-01-29
!references 83-00143, 83-00034
!topic Lifetime of a temporary file and its name

!summary 84-11-28

1) An implementation is allowed to delete a temporary file immediately after closing it.

2) The NAME function is allowed to raise USE_ERROR if its argument is associated with an external file that has no name, in particular, a temporary file.

!question 84-11-28

1) The wording for CREATE says that a temporary file is "an external file that is not accessible after the completion of the main program." Does this wording imply an upper limit to the lifetime of the temporary file, or can an implementation delete a temporary file as soon as the file is closed?

2) If an implementation does not normally give names to temporary files, what does the NAME function return when its argument is associated with a temporary file?

!response 84-11-28

The effect of calling CLOSE is to sever the association between a file declared in an Ada program and an external file [14.2.1(9)]. After this association is broken, the status of the external file is not specified by the Standard. In particular, its continued existence is not guaranteed. For example, such an external file might be deleted by some other program or activity before an attempt is made to open it again. Since a temporary file is an external file, temporary files need not be retained after they are closed (see example below).

In some implementations, when a temporary file is created, it is not given a name that can be used in a subsequent OPEN operation. In such a case, the NAME function is allowed to raise USE_ERROR. ("The exception USE_ERROR is raised if an operation is attempted that is not possible for reasons that depend on characteristics of the external file" [14.4(5)].)

The following example illustrates these points:

```
CREATE (FILE, OUT_MODE, "");
-- write some data
declare
     FILE_NAME := constant STRING := NAME (FILE);
     -- could raise USE_ERROR
begin
     CLOSE (FILE);
     -- file could now be deleted
     OPEN (FILE, IN_MODE, FILE_NAME);
     -- NAME_ERROR raised if the file no longer exists
end;
```

Ada Commentary ai-00047-bi.wj downloaded on Mon May 9 16:50:02 EDT 1988

!standard 14.03.01 (04) 87-09-12 AI-00047/08
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 87-07-30 (corrected in accordance with AI-00486)
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (7-0-1) 86-02-20
!status work-item 86-01-23
!status received 84-01-29
!references AI-00486, 83-00095, 83-00096, 83-00097, 83-00824
!topic Effect of RESET on line and page length

!summary 86-03-05

Calling RESET resets the line and page lengths to UNBOUNDED.

!question 87-09-09

14.3.1(4) states:

> For the procedure RESET: If the file has the current mode OUT_FILE, has the effect of calling NEW_PAGE, unless the current page is already terminated; then outputs a file terminator. If the new file mode is OUT_FILE, the page and line lengths are unbounded. For all modes, the current column, line, and page numbers are set to one.

Consider the following example:

```
CREATE (FT, OUT_FILE);
SET_PAGE_LENGTH (FT, 4);
RESET (FT, OUT_FILE);  -- (1)
                       -- (2)
```

The RESET at (1) did not change the mode of the file. Does PAGE_LENGTH(FT) equal 0 or 4 at (2)?

!recommendation 86-03-05

Calling RESET resets the line and page lengths to UNBOUNDED.

!discussion 86-05-20

14.3.1(4) says:

> If the new file mode is OUT_FILE, the page and line lengths are unbounded.

The word "new" is present simply to indicate the mode after the RESET is executed; it was not meant to imply the page and line lengths are reset only

if the mode has changed to OUT_FILE from another mode. Resetting with mode OUT_FILE, explicitly or by default, always causes the page and line lengths to be set to zero.

Ada Commentary ai-00048-bi.wj downloaded on Mon May 9 17:45:01 EDT 1988

!standard 14.03.02 (01) 87-02-23 AI-00048/12
!standard 14.03.01 (05)
!class binding interpretation 86-02-21
!status approved by WG9/AJPO 87-02-20
!status approved by Director, AJPO 87-02-20
!status approved by Ada Board (18-0-3) 87-02-19
!status approved by WG9 86-05-09
!status committee-approved 86-02-21 (8-0-0)
!status reconsidered by committee 86-02-20 (7-0-1)
!status committee-approved (8-2-0) 85-11-20 (pending editorial review)
!status work-item 85-03-04
!status returned to committee by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-06-29
!status board returned to committee 84-06-29
!status committee-approved 84-02-06
!status work-item 84-01-29
!references 83-00033, 83-00355, 83-00362, 83-00404, 83-00495, 83-00703
!topic Default files can be closed, deleted, and re-opened

!summary 86-03-05

The CLOSE operation can be applied to a file object that is also serving as the default input or default output file. The effect is to close the default file. A subsequent OPEN operation can have the effect of opening the default file as well. Similarly, a DELETE operation can be applied to a file object that is serving as the default file.

The exception MODE_ERROR is raised by OPEN if the specified mode is OUT_FILE and the file object being opened is serving as the default input file. Similarly, MODE_ERROR is raised if the specified mode is IN_FILE and the file object being opened is serving as the default output file.

!question 85-01-02

Although it is not possible to use RESET to change the mode of a file that is serving as a default input or output file, is it possible to close the file and reopen it with a different mode? Is it possible to delete the current default file?

!recommendation 86-04-02

The CLOSE operation can be applied to a file object that is also serving as the default input or default output file. The effect is to close the default file. A subsequent OPEN operation can have the effect of opening the default file as well. Similarly, a DELETE operation can be applied to a file object that is serving as the default file.

The exception MODE_ERROR is raised by OPEN if the specified mode is OUT_FILE and the file object being opened is serving as the default input file. Similarly, MODE_ERROR is raised if the specified mode is IN_FILE and the file object being opened is serving as the default output file.

!discussion 86-04-02

Consider the following:

```
declare
    F : TEXT_IO.FILE_TYPE;
begin
    OPEN (F, IN_FILE, "FOO");
    SET_INPUT (F);
    CLOSE (F);                          -- is the default file closed?

    OPEN (F, OUT_FILE, "BAR");          -- what is the default file now?
    GET (CURRENT_INPUT, ...);           -- is there an exception?
end;
```

Is the close operation on F permitted even though F is serving as the default input file? If so, is the default file also closed, or is it still associated with external file "FOO"? If the open operation is successful, is the default input file associated with external file "BAR" with mode OUT_FILE? Is the default file associated with any external file after the OPEN operation?

To answer these questions, we need to consider several definitions given in the Standard. 14.1(2) says:

> Input and output operations are expressed as operations on objects of some FILE TYPE, rather than directly in terms of the external files. In the remainder of this chapter, the term FILE is always used to refer to a file object; the term EXTERNAL FILE is used otherwise.

To make the discussion clearer here, when quoting from the Standard, we will substitute the term "file object" when the Standard uses just the term "file."

14.1(6) goes on to describe the conceptual model underlying the I/O operations:

> Before input or output operations can be performed on a [file object], the [file object] must first be associated with an external file. While such an association is in effect, the [file object] is said to be OPEN, and otherwise the [file object] is said to be CLOSED.

In other words, the effect of executing an open operation is just to establish an association between a file object and an external file. (Of course, while this association is in effect, operations performed on the file object will affect (and be affected by) the external file.)

The semantics of the open operation for text files are given in 14.3.1(1-2) and 14.2.1(6), which says:

> [OPEN] associates the given [file object] with an existing external file having the given name and form, and sets the current mode of the given [file object] to the given mode. The given [file object] is left open.

The definition of the close operation is given in 14.3.1(1, 3) and 14.2.1(9), which says:

> [CLOSE] severs the association between the given [file object] and its associated external file. The given [file object] is left closed.

Finally, the definition for SET_INPUT says [14.3.2(3)]:

> [SET_INPUT] sets the current default input [file object] to FILE.

The effect of SET_INPUT (and similarly, of course, SET_OUTPUT) is to make the default input file object identical with the file object that is the value of SET_INPUT's argument, i.e., the default input file is not independently associated with any external file, but rather shares the file object that is specified by SET_INPUT. After executing SET_INPUT(F), CURRENT_INPUT and F have the same file object value. This means operations on F and operations on the default input file have exactly the same effect, e.g., NEW_LINE(F) changes the line count for the default input file.

When the operation CLOSE(F) is executed, the association between the shared file object and the external file is severed, but there is no implication that the file object serving as the default input file somehow loses its association with the file object that is the value of F. Instead, the effect is simply to sever the external file's association with the file object associated with both F and CURRENT_INPUT. A similar effect occurs for DELETE. DELETE(F) deletes the associated external file and also severs the association between the file object named by F (which is also the default file object) and the external file.

A subsequent open operation that uses F's file object establishes an association between this file object and an external file. Since this file object is still serving as the default input file object, the association is established for CURRENT_INPUT as well. If the OPEN operation in the example had specified mode IN_FILE, then the subsequent GET operation would apply to the external file "BAR". The effect would be the same as if file object F had been named in the argument instead of CURRENT_INPUT.

Of course, the OPEN operation in the example attempts to establish an OUT_FILE association between F's file object and the external file. This poses a problem since F's file object is also the current default input file object and the Standard, in general, tries to ensure that the mode of a default file object is consistent with its use as an input or output file. In particular, SET_INPUT and SET_OUTPUT raise MODE_ERROR if the mode of the file object being established as a default file is not consistent with the file's use for input or output [14.3.2(4, 7)]. Similarly, RESET raises

MODE_ERROR upon an attempt to change the mode of a file that is either the current default input file object or the current default output file object [14.3.1(5)]. To ensure a default file object has an appropriate mode, MODE_ERROR should also be raised by OPEN if the file object being associated with an external file is serving as a default file object and the mode specified by OPEN is inconsistent with the file object's use as a default input or output file. (So in the example, MODE_ERROR should be raised by OPEN since the file object denoted by F is serving as the default input file object.)

An objection to allowing a default file object to be closed and subsequently reopened is that it forces a somewhat more complicated implementation. Suppose an implementation wants to represent a file object as a record. It can't use a simple record representation because SET_INPUT and SET_OUTPUT require that file objects be shared. So the implementation must use a pointer to a record. Given that pointers must be used, it would be convenient to represent a closed file with a null pointer, e.g., when F's declaration is elaborated, its real value is set to null. The open operation allocates and initializes a record object (as well as gaining access to the external file). The close operation releases access to the external file and sets the pointer to null (e.g., CLOSE (F) would set F's value to null). Such an implementation could release the storage used by the designated record object if this object were not designated by any other pointer, which would be the case if F's file object was not also being used as a default file object. When F's file object is also a default file object, the close operation cannot assign null to F because a subsequent open operation for F must also establish an external file association for the default file object, i.e., the designated record object used for F's new external file association must still be the same as the default file object.

In short, a somewhat more efficient implementation would be possible if closing a default file object implied the default file object was no longer shared with the value of any file variable. But the Standard does not allow such an interpretation (and the additional implementation burden is not sufficiently great to warrant considering such an interpretation further.)

Ada Commentary ai-00050-bi.wj downloaded on Mon May 9 17:45:03 EDT 1988

!standard 14.03.06 (13) 86-12-01 AI-00050/11
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status WG9/ADA Board approved (provisional) 85-05-13
!status committee-approved (10-1-1) 85-05 (by letter ballot)
!status committee-approved (4-0-5) 85-02-25 (pending letter ballot)
!status work-item 84-11-05
!status received 84-01-29
!references 83-00105, 83-00106, 83-00107, 83-00393, 83-00475, 83-00669, 83-00707
!topic When does GET_LINE call SKIP_LINE?

!summary 85-03-11

GET_LINE reads characters until either the end of the string is met or until END_OF_LINE is true. If the end of the string has been met, SKIP_LINE is not called even if END_OF_LINE is true. In particular, no characters are read if the string is null.

!question 84-10-30

If a file is positioned just before a line terminator and GET_LINE is called with a null string variable, is the line terminator skipped? If the string variable has length N and exactly N characters remain to be read on the current line, is the line terminator skipped after reading the N characters?

Similarly, if a file is positioned just before the file terminator (e.g., just after having skipped the last page terminator in a file) and GET_LINE is called with a null string variable, is END_ERROR raised or is there no effect?

!recommendation 85-03-11

GET_LINE reads characters until either the end of the string is met or until END_OF_LINE is true. SKIP_LINE is called if and only if END_OF_LINE is true and the end of the string has not been met.

!discussion 85-10-23

The second sentence of this paragraph specifies two conditions under which GET_LINE stops reading:

"the end of the line is met"

"the end of the string is met"

Both conditions can be true at the same time, e.g., if GET_LINE is called with a null string variable and the file is positioned just before a line terminator. Specifically, the Standard says:

> "Reading stops if the end of the line is met, in which case the procedure SKIP_LINE is then called ...; reading also stops if the end of the string is met."

If the phrase "end of the line is met" is understood to mean "met while attempting to read another character because the end of the string has not yet been met," then if GET_LINE is called with a variable of length N and there are N characters left to read, these characters will be read and SKIP_LINE will not be called.

If the phrase "end of the line is met" is understood to mean END_OF_LINE is true after reading has stopped (in particular, after reading has stopped because no characters can be added to the output string), then SKIP_LINE will be called if the number of characters to be read equals the number left on the line.

To choose between these interpretations requires a careful consideration of the wording together with a consideration of the intent. The rule says: "Reading stops if the end of the line is met, IN WHICH CASE the procedure SKIP_LINE is called ...". In other words, there are other cases when SKIP_LINE is NOT invoked. The description then goes on: "...; reading also stops if the end of the string is met." Here is the other case that we were expecting to be described, suitably separated by a semicolon. Hence, if the end of the string is encountered, reading stops immediately, and no test is made for the end of the line; no invocation of SKIP_LINE occurs, whether or not END_OF_LINE is true.

Another way of reaching the same conclusion is to interpret the phrase "end of the line is met" to mean that an attempt has been made to read PAST the end of the line. This does not include the case where the end of the string has been met, since you can only "meet" the end of the line while attempting to read a character.

The intent behind GET_LINE was to allow successive lines of a file to be read. Suppose one is reading fixed length records, e.g., records of length 80 using a string variable of length 80. Under the proposed interpretation, SKIP_LINE will have to be called after each GET_LINE call, or else every other call to GET_LINE will return a null string. Alternatively, GET_LINE should be called with a string of length 81, in which case, successive calls to GET_LINE will suffice to read successive records. Either approach allows lines to be read conveniently.

Since on balance, the wording appears to specify that SKIP_LINE is called when only one of the two reasons for stopping is satisfied and since this interpretation allows the intent to be achieved, the recommended interpretation is:

> GET_LINE reads characters until either the end of the string is met or until END_OF_LINE is true. SKIP_LINE is called only if the end of the string has not been met and END_OF_LINE is true.

Given this interpretation, if the length of the string is N and there are exactly N characters remaining on the line before END_OF_LINE is true, then SKIP_LINE is not called after reading the N characters. This holds even if the string is a null string (so N is zero) and the file is positioned directly before a line terminator. When positioned directly in front of the file terminator and called with a null string, END_ERROR is not raised since SKIP_LINE is not called.

Ada Commentary ai-00051-bi.wj downloaded on Mon May 9 17:50:01 EDT 1988

!standard 14.03.07 (06) 86-07-23 AI-00051/07
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 84-01-29
!references 83-00091, 83-00092, 83-00094, 83-00206, 83-00407, 83-00412, 83-00491
!topic Reading "integer literals"

!summary 84-05-11

Integer GET reads according to the syntax of an optionally signed numeric literal that does not contain a point. It raises DATA_ERROR if the characters read do not form a legal integer literal. For example, if integer GET attempts to read 0.3, 0E-3, or 20#0#, reading stops before the decimal point for 0.3, after the 3 for 0E-3, and after the second # for 20#0#; DATA_ERROR is raised for 0E-3 since legal integer literals are not allowed to have exponents containing minus signs. DATA_ERROR is also raised for 20#0#, since 20 is not an allowed base value.

!question 84-07-13

The Standard says GET "reads according to the syntax of an integer literal," and raises DATA_ERROR if the input sequence "does not have the required syntax." Does reading stop at the minus sign for 0E-3 or at the 3? If it stops at the 3, is DATA_ERROR raised? For 0.3, does reading stop at the 3 or at the point?

!recommendation 84-05-08

If the character sequence being read by integer GET satisfies the syntax for numeric_literal (preceded by an optional sign) and has a point, reading stops just before the point. DATA_ERROR is raised if the characters read do not satisfy the syntax for numeric_literal. If a character sequence satisfies the syntax for a numeric literal without a point but the literal fails to satisfy other legality rules, then GET raises DATA_ERROR.

!discussion 84-05-11

The Standard says:

> "[GET] reads according to the syntax of an integer literal (which may be a based literal)." [14.3.7(6)]
>
> "The exception DATA_ERROR is raised if the sequence input does not have the required syntax or if the value obtained is not of the subtype NUM." [14.3.7(8)]

> "an integer literal is a numeric literal without a point" [2.4(1)], and numeric_literal is defined by a context-free syntax rule [2.4(2)].
>
> "An exponent for an integer literal must not have a minus sign" [2.4.1(4)].
>
> "The base [of a based literal] must be at least two and at most sixteen" [2.4.2(1)].

Now consider the examples:

```
0.3
2#0.3#
0E-3
2#0#E-3
20#0#
```

There are two issues raised by the current wording -- when does reading stop and when is DATA_ERROR raised?

The question of when reading stops is answered by the definition in 2.4(1). This narrative definition specifies the syntactic form of integer literals, i.e., they have the form of numeric_literals without a point. Hence, when reading 0.3, reading stops when the point is encountered and GET returns the value zero; the next character to be read is '.'.

The intent in 14.3.7(8) was to ensure that GET consider a sequence of characters to form an integer literal if and only if the same sequence would be allowed in an Ada program as an integer literal. The curent wording does not satisfy this intent because it requires only that the input sequence have the required "syntax". The intent was that DATA_ERROR should be raised if the characters read do not form a LEGAL integer literal.

For example, 2#0#E-3 has the form of a based literal without a point, and so satisfies the syntactic requirements for an integer literal. It is not, however, a legal integer literal because of the minus sign in the exponent. The syntactic rule for integer literals requires that reading stop after the 3. The legality rules require that DATA_ERROR then be raised. Similarly, for 20#0#, reading stops after the second # but DATA_ERROR is raised because 20 is not in the allowed range of base values.

When reading 2#0.3#, reading stops when the point is encountered, but since 2#0 is not a legal integer literal (it does not satisfy the syntax for decimal_literal or based_literal), DATA_ERROR is raised.

Ada Commentary ai-00103-ra.wj downloaded on Mon May 9 17:50:02 EDT 1988

!standard 04.10 (04) 86-07-23 AI-00103/06
!class ramification 85-09-04
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-04
!status work-item 84-06-11
!status received 83-11-07
!references 83-00154, 83-00659, 83-00665
!topic Accuracy of a relation between two static universal real operands

!summary 85-09-17

The relational and membership operations for static universal real operands must be evaluated exactly.

!question 85-09-17

The last sentence of section 4.10(4) says "if a universal expression is a static expression, then the evaluation must be exact". However, no corresponding rule exists for the evaluation of a static expression such as

```
0.1 = 0.01E1
```

which is intended always to deliver the result TRUE. This is because an expression with result type BOOLEAN is not classed as a universal expression. Must static relational and membership expressions with universal real operands be evaluated exactly?

!response 85-10-17

The Standard defines the accuracy of relational and membership operations in terms of model numbers of the type (see 4.5.7(10,11)). According to 3.5.6(3), a set of model numbers is associated with every real type, and 3.5.6(5) says universal real is a real type, so there are model numbers for the type universal real. The Standard does not specify the form of the model numbers (e.g., do they have the form specified in 3.5.7(4) with infinite mantissa, or the form given by 3.5.9(4) with an appropriate 'SMALL?), but regardless of the form that is considered to be used, it does seem clear from 4.10(4) that the model interval for a static universal real expression is a point, and so relational and membership operations for static universal real operands are to be evaluated exactly, as required by 4.5.7(10) when comparing model numbers of a type.

Ada Commentary ai-00120-co.wj downloaded on Mon May 9 18:45:01 EDT 1988

!standard 08.07 (03) 86-12-01 AI-00120/05
!class confirmation 86-05-12
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (9-0-0) 86-05-12 (pending editorial review)
!status work-item 86-01-22
!status received 83-11-07
!references 83-00172
!topic Overload resolution for assignment statements

!summary 86-01-23

The type of the right-hand side of an assignment statement can be used to determine an overload resolution of the left-hand side.

!question 86-01-22

May the type of the right-hand side of an assignment statement be determined and used to help in overload resolution for the left-hand side?

!response 86-04-15

The type of the right-hand side of an assignment statement can be used to determine an overload resolution of the left-hand side. Consider the example given below:

```
declare
    type R is
        record
            A : INTEGER;
        end record;

    type Q is
        record
            A : BOOLEAN;
        end record;

    type S is access R;
    type V is access Q;

    function F return S;  --  Fone
    function F return V;  --  Ftwo

    X : INTEGER := 5;

begin

    F.A := X;

end;
```

Each statement is a complete context (8.7(3)). Hence the syntax rules, the scope and visibility rules, and the rules (a) -- (f) of 8.7 govern the possible interpretations.

It is a consequence of the scope and visibility rules that X in F.A := X is of type INTEGER. According to rule (a) and the requirement in paragraph 5.2(1), F.A must also be of type INTEGER. Since F.A is a selected component (syntax rules), A is of type INTEGER. Therefore A is a component of an object of type R because only this component with name A is of type integer (scope and visibility rules).

The rule stated in paragraph 4.1.3(6) (applicable because of rule (c)) now tells us that the prefix F is appropriate for the type R; that is either

- the type of the prefix is R

or

- the type of the prefix is an access type whose designated type is R.

Hence, the result type must be an access type whose designated type is R and consequently the result type is S. Finally, the rules for the resolution of overloaded subprogram calls (applicable because of rule (f)) provide that F is Fone.

Ada Commentary ai-00128-bi.wj downloaded on Mon May 9 18:45:02 EDT 1988

!standard 04.09 (02) 86-07-23 AI-00128/04
!class binding interpretation 83-11-07
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 83-11-07
!references 83-00180, 83-00242
!topic No membership tests or short-circuit operations in static expressions

!summary 84-07-13

Membership tests and short-circuit control forms are not allowed in static expressions because neither of these are operators.

!question 84-07-13

The expression

```
10 in Some_Subtype
```

is a static expression according to the current wording, even if Some_Subtype is not static. Is it intended that this expression be considered static if Some_Subtype is static?

!recommendation 84-03-14

The use of a membership test or short-circuit control form makes an expression non-static.

!discussion 84-03-14

The expression

```
10 in Some_Subtype
```

is a static expression according to 4.9(2) since 10 is a primary having one of the permitted forms, "in" is not an operator (it is an operation; see 4.5(1,2)), and Some_Subtype is neither an operator nor a primary. This conclusion is not acceptable, however, since Some_Subtype need not be a static subtype, and if it is not static, the value of the expression cannot be determined at compile-time, as the language sometimes requires. For example, the following type declaration is illegal if the value given for digits exceeds SYSTEM.MAX_DIGITS:

```
type T is digits BOOLEAN'POS(10 in Some_Subtype)+5;
```

Since the value of 10 in Some_Subtype cannot necessarily be determined prior to execution of the main program, the legality of the type declaration cannot be determined.

The current wording says, in part, that "an expression of a scalar type is said to be static ... only if .. every operator denotes a predefined operator." This wording is intended to say that the only operations permitted in static expressions are those denoted by operators. 10 in Some_Subtype is not a static expression because a membership test is not an operator -- it is an operation (see 4.5(3)). Because static expressions are only supposed to contain operators, the short-circuit control operations are also not allowed in static expressions.

Ada Commentary ai-00132-bi.wj downloaded on Mon May 9 18:50:01 EDT 1988

!standard 13.04 (07) 86-07-23 AI-00132/05
!class binding interpretation 83-11-07
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9-0-0) 85-11-20
!status work-item 85-04-08
!status received 83-11-07
!references 83-00185
!topic Static constraints and component clauses

!summary 85-04-08

A record component clause is only allowed for a record component having a constraint if the constraint is static and, if the component has subcomponents that are constrained, each subcomponent constraint is static.

!question 85-04-08

For record representation clauses, 13.4(7) says, "A component clause is only allowed for a component if any constraint on this component or on any of its subcomponents is static." This wording (incorrectly) allows a record representation clause for a component even if a subcomponent has a nonstatic constraint.

!recommendation 86-01-20

A component clause is only allowed for a record component if the only constraints (if any) upon the component or its subcomponents are all static constraints.

!discussion 85-12-18

It was intended to allow record representation clauses only for components whose size was determinable at compile time, i.e., it was intended to require that any constraint imposed on a component be static, and for those components that had subcomponents (namely, arrays and records), that any constraint specified for each subcomponent similarly be static. (Note that a component or subcomponent has only one constraint, which may consist of several expressions.) The current wording does not state the intended requirement, since it allows a component clause to be specified as long as at least one subcomponent has a static constraint.

Ada Commentary ai-00137-ra.wj downloaded on Mon May 9 18:50:02 EDT 1988

!standard 04.05.06 (06) 86-07-23 AI-00137/05
!class ramification 84-02-06
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 83-11-07
!references 83-00192, 83-00312
!topic Exponentiation with floating point operand

!summary 84-03-16

Since the model interval for X*X*X*X is sometimes smaller than the interval for (X*X)*(X*X), an implementation cannot compute X**4 as sqr(sqr(X)), where sqr(Y) computes Y*Y. In general, exponentation to the Nth power must be implemented using N-1 multiplications to ensure the required accuracy is obtained.

!question 84-08-29

Can fewer than N-1 multiplications be used in computing X**N (for N > 0) even though the Standard defines the model interval of the result in terms of repeated multiplications?

!response 84-08-29

Any method can be used that provides results within the limits permitted by 4.5.7(9). In general, however, the error bounds for X*X*X*X can be smaller than the bounds for (X**2)**2, so the result of (X**2)**2 can lie outside the bounds for X*X*X*X for some values of X. Consequently, the only method guaranteed to stay within the error bounds specified by 4.5.7(9) is to perform N-1 multiplications when N > 0.

In practice, performing N-1 multiplications is not inefficient, because 90% of the time the exponent is 2, 7% of the time, it is 3, and only in 3% of the cases is the exponent larger than three [Wichmann, B. A., "Algol 60 Compilation and Assessment," Academic Press, 1973]. So the possibility of reducing the number of multiplications does not arise very often.

It should also be noted that X**N for negative N should not be computed as (1.0/X)**(abs N). For example, if X is 3.0 and N is 3, 81.0 is a model number and 1.0/81.0 must lie within the smallest possible model interval. But 1.0/3.0 is not a model number and so is not exactly representable as a binary floating point value. The accumulated error in multiplying an approximation to 1/3 by itself will yield a result outside the model interval required by Ada if the exponent is made sufficiently large.

Ada Commentary ai-00138-bi.wj downloaded on Tue May 10 08:45:02 EDT 1988

!standard 03.04 (10)
!standard 03.04 (22)
!standard 13.01 (03)
!standard 13.06 (01)
!class binding interpretation 83-11-07
!status approved by WG9/AJPO 87-02-20
!status approved by Director, AJPO 87-02-20
!status approved by Ada Board (21-0-0) 87-02-19
!status approved by WG9 86-05-09
!status committee-approved (6+2-2-2) 86-02-20 (by ballot)
!status committee-approved 85-11-22 (7-0-1)
!status work-item 84-06-12
!status received 83-11-07
!references AI-00099, 83-00195, 83-00196, 83-00652
!topic Representation clauses for derived types

!summary 86-05-20

If an aspect of a parent type's representation has been specified by an implicit or explicit representation clause and no explicit representation clause is given for the same aspect of the derived type, the representation of the derived and parent types are the same with respect to this aspect.

An explicit length clause for STORAGE_SIZE of a task type, for SIZE (of any type), or for SMALL of a fixed point type, an explicit enumeration representation clause, an explicit record representation clause, or an explicit address clause for a task type can be given for a derived type (prior to a forcing occurrence for the type) even if a representation clause has also been given (explicitly or implicitly) for the same aspect of the parent type's representation. (But only a length clause is allowed for a derived type if the parent type has derivable subprograms.)

An expression in an implicit representation clause is not evaluated when the implicit clause is elaborated.

!question 85-12-29

3.4(10) states:

> If an explicit representation clause exists for the parent type and if this clause appears before the derived type definition, then there is a corresponding representation clause (an implicit one) for the derived type.

This wording raises three questions:

> If the only representation clause for a parent type is an implicit clause, does a representation clause also exist for the derived type?
>
> Can an implicit representation clause be overridden by an explicit one?

If the parent's representation clause contains a non-static expression (e.g., for 'STORAGE_SIZE), is the expression re-evaluated for each implicit representation clause?

!recommendation 85-12-29

If an aspect of a parent type's representation has been specified by an implicit or explicit representation clause and no explicit representation clause is given for the same aspect of the derived type, the representation of the derived and parent types are the same with respect to this aspect.

Except for an explicit or implicit representation clause specifying STORAGE_SIZE for an access type or SMALL for a fixed point type, the existence of a representation clause (explicit or implicit) for some aspect of a parent type's representation does not affect the legality of an explicit representation clause for the same aspect of the derived type.

An expression in an implicit representation clause is not evaluated when the implicit clause is elaborated.

!discussion 86-06-19

Consider the following chain of derived types:

```
type A is (A1, A2, A3);
for A use (1, 2, 4);     -- explicit representation clause
for A'SIZE use 8;        -- explicit representation clause

type B is new A;
-- implicit representation clauses for B, by 3.4(10)
```

Paragraph 13.6(1) states that each type can have only one representation clause for a given aspect of the type. It is not intended that this restriction (or the similar ones given in 13.1(3)) apply to an implicit representation clause created by a derived type declaration. Hence, a new representation clause can be given for B.

Now suppose there is also the declaration:

```
type C is new B;
```

3.4(10) does not specify that a representation clause exists for C since C's parent type only has implicit representation clauses, but the intent is that a derived type have the same representation as its parent type unless an explicit representation clause specifies a different representation. Hence, C'SIZE = B'SIZE = A'SIZE = 8.

Expressions in an implicit representation clause are not evaluated again. The intended effect of an implicit representation clause is just to ensure that a derived type and its parent type have the same representation. Thus, in an example like the following, function F is only invoked once:

```
        function F return INTEGER is
        begin ... end;

        task type TSK is
             ...
        end TSK;
        for TSK'STORAGE_SIZE use F;

        type NEW_TSK is new TSK;
-- for NEW_TSK'STORAGE_SIZE use F; -- implicit clause here
```

NEW_TSK'STORAGE_SIZE must have the same value as TSK'STORAGE_SIZE, and F is only invoked when the explicit representation clause is elaborated.

Ada Commentary ai-00139-ra.wj downloaded on Tue May 10 08:45:03 EDT 1988

!standard 07.04.02 (07) 86-07-23 AI-00139/04
!class ramification 83-11-07
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 83-11-07
!references 83-00197
!topic The declaration of "additional operations" for access types

!summary 84-03-26

A consequence of the rule in 7.4.2(7, 8) is that an object A1 can be declared such that the name A1(1) is illegal when A1.all(1) would be legal. Similar examples exist for slices, for the attributes 'FIRST, 'LAST, 'LENGTH, and 'RANGE, and for selection of a record component (e.g., R1.C1 can be illegal when R1.all.C1 is legal).

!question 84-07-13

Version 1.2 of the validation suite contains the following test:

```
-- B74205B-B.ADA

-- CHECK THAT FOR AN ACCESS TYPE WHOSE DESIGNATED TYPE IS A PRIVATE TYPE
-- ADDITIONAL OPERATIONS FOR THE ACCESS TYPE WHICH DEPEND ON
-- CHARACTERISTICS OF THE FULL DECLARATION OF THE PRIVATE TYPE ARE NOT
-- DECLARED BEFORE THE EARLIEST PLACE WITHIN THE IMMEDIATE SCOPE OF THE
-- ACCESS TYPE DECLARATION AND AFTER THE FULL DECLARATION OF THE PRIVATE
-- TYPE.

-- (1) CHECK FOR COMPONENT SELECTION WITH RECORD TYPES
-- (2) CHECK FOR INDEXED COMPONENTS AND SLICES WITH ARRAY TYPES
-- (3) CHECK FOR USE OF 'FIRST, 'LAST, 'RANGE, AND 'LENGTH WITH ARRAY
--     TYPES

-- DSJ 5/2/83

PROCEDURE B74205B IS

     PACKAGE PACK1 IS
          TYPE T1 IS PRIVATE ;
          TYPE T2 IS PRIVATE ;
          PACKAGE PACK2 IS
               TYPE ACC1 IS ACCESS T1 ;
               TYPE ACC2 IS ACCESS T2 ;
          END PACK2 ;
     PRIVATE
          TYPE T1 IS ARRAY ( 1 .. 2 ) OF INTEGER ;
```

```
        TYPE T2 IS
             RECORD
                  C1, C2 : INTEGER ;
             END RECORD ;
   END PACK1 ;

   PACKAGE BODY PACK1 IS
        A1 : PACK2.ACC1 := NEW T1'(2,4) ;   -- LEGAL
        A2 : PACK2.ACC1 := NEW T1'(6,8) ;   -- LEGAL
        R1 : PACK2.ACC2 := NEW T2'(3,5) ;   -- LEGAL
        R2 : PACK2.ACC2 := NEW T2'(7,9) ;   -- LEGAL

        X1 : INTEGER := A1(1) ;             -- ERROR: A1(1)
        X2 : INTEGER := A1'FIRST ;          -- ERROR: A1'FIRST
        X3 : INTEGER := A1'LAST ;           -- ERROR: A1'LAST
        X4 : INTEGER := A1'LENGTH ;         -- ERROR: A1'LENGTH
        B1 : BOOLEAN := 3 IN A1'RANGE ;     -- ERROR: A1'RANGE

        X5 : INTEGER := R1.C1 ;             -- ERROR: R1.C1

   END PACK1 ;

BEGIN

   NULL ;

END B74205B ;
```

Tests B74205C and B74205D are similar tests where type T1 is replaced by an incomplete type.

Question 1: On what basis are the initializations of A1, A2, R1 and R2 considered to be legal?

Question 2: Suppose the following text were added in the above test immediately following the declaration of X5:

```
   XX1 : INTEGER := A1.all(1);              -- Legal?
   XX2 : INTEGER := A1.all'FIRST;           -- Legal?
   XX3 : INTEGER := A1.all'LAST;            -- Legal?
   XX4 : INTEGER := A1.all'LENGTH;          -- Legal?
   XB1 : BOOLEAN := 3 in A1.all'RANGE;      -- Legal?
   XX5 : INTEGER := R1.all.C1;              -- Legal?
```

These differ from the declarations for X1, X2, X3, X4, B1 and X5 above only by adding an explicit .all before applying indexing, attributes, and selection. Are all of these declarations legal? Not legal?

!response 84-03-26

1. Allocator operations are declared for ACC1 and ACC2 in PACK2 [3.8.2(1) and 3.3.3(2)]. Since these are basic operations, they are visible in the

package body for PACK1 [8.3(18) and 8.2(3)]. Qualification is a basic operation for T1 and is declared in PACK1. Finally, the aggregate formation operations for T1 and T2 are declared in the private part of PACK1, and so are visible in the package body. The initializations of A1, A2, R1, and R2 are legal since all the required operations are visible.

2. The examples would all be legal since .all is declared in the visible part of PACK2 [3.8.2(1)], and all the other operations are only declared for T1 and T2 after their full declarations in the private part of PACK1. A1(1) is not legal, for example, since the index selection operation for ACC1 is only declared within the immediate scope of ACC1 [7.4.2(8)], i.e., only within PACK2's body (if it existed), and hence is not visible at the points indicated by the test.

Ada Commentary ai-00143-ra.wj downloaded on Tue May 10 08:50:01 EDT 1988

!standard 03.05.09 (06) 86-07-23 AI-00143/04
!class ramification 84-01-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-02-06
!status received 84-01-10
!references 83-00234
!topic Model numbers for delta 1.0 range -7.0 .. 8.0

!summary 84-03-16

Given

```
type F is delta 1.0 range -7.0 .. 8.0;
```

The model numbers for F do not include the value 8.0 and F'MANTISSA must be 3.

!question 84-01-10

Consider the model numbers for F:

```
type F is delta 1.0 range -7.0 .. 8.0;
```

The wording requires that F'MANTISSA be the SMALLEST integer number for which each bound of the specified range is either a model number or lies at most small distant from a model number. This means F'MANTISSA is required to be 3 since the range -7.0 .. 7.0 fits in 3 signed bits, and 8 is "at most" 1.0 from a model number, namely, 7. Is this analysis correct? Note that this implies the upper bound of the range is not represented as a model number.

!response 84-03-17

The analysis is correct. The upper and lower bounds for a fixed point type can lie outside the range of model numbers.

Ada Commentary ai-00144-bi.wj downloaded on Tue May 10 08:50:02 EDT 1988

!standard 03.05.09 (09) 87-09-12 AI-00144/10
!class binding interpretation 84-01-10
!status approved by WG9/AJPO 87-07-30 (corrected in accordance with AI-00471)
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-20
!status work-item 85-11-18
!status returned to committed by WG9 85-11-18
!status committee-approved (8-0-1) 85-09-04
!status work-item 84-06-12
!status received 84-01-10
!references AI-00143, AI-00471, 83-00235, 83-00822
!topic A fixed point type declaration cannot raise an exception

!summary 87-09-09

A fixed point type declaration cannot raise an exception. Declarations such as:

```
type F is delta 2.0**(-15) range -1.0 .. 1.0;
```

are legal even if F'SIZE is equal to 16 so 1.0 is not a representable value.

!question 87-09-09

The equivalence stated in 3.5.9(9) suggests that the declaration:

```
type F is delta 2.0**(-15) range -1.0 .. 1.0;
```

will raise NUMERIC_ERROR if the implementation chooses a predefined type such that F'SIZE = 16, i.e., such that 1.0 is not a representable value. Is this correct, or is the equivalence only for expository purposes, implying that NUMERIC_ERROR cannot be raised?

!recommendation 87-09-09

A fixed point type declaration cannot raise an exception. Declarations such as:

```
type F is delta 2.0**(-15) range -1.0 .. 1.0;
```

are legal even if F'SIZE is equal to 16.

!discussion 87-09-09

The declaration:

```
type F is delta 2.0**(-15) range -1.0 .. 1.0;
```

is stated to be equivalent to:

```
type %FP is new %PDF;
subtype F is %FP range %FP(-1.0) .. %FP(1.0);
```

An implementation is allowed to choose the predefined type %PDF such that all and only the model numbers of F are representable, e.g., %PDF might occupy just 16 bits (see AI-00143). In such a case, the upper bound of %FP, the base type of F, is 1.0-2.0**(-15), i.e., the value 1.0 lies just outside the range of the base type. A conversion of a value that lies outside the target type's base type is allowed to raise NUMERIC_ERROR [4.5.7(7)], so it would seem that the conversion, %FP(1.0) is allowed (or required) to raise NUMERIC_ERROR. However, it was intended that declarations like the one given be accepted for a base type that occupies just 16 bits. It was intended to allow programmers to write:

```
type F is delta 2.0**(-15) range -1.0 .. 1.0;
```

rather than requiring programmers to write:

```
type F is delta 2.0**(-15) range -1.0 .. 1.0-2**(-15);
```

In short, the declaration given in the question is legal and neither NUMERIC_ERROR nor any other exception can be raised by such a declaration. (Of course, even if the type declaration is accepted, NUMERIC_ERROR could be raised for F'(1.0).)

Ada Commentary ai-00145-ra.wj downloaded on Thu May 12 16:45:03 EDT 1988

!standard 03.05.09 (14) 86-07-23 AI-00145/04
!class ramification 84-01-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 84-01-10
!references 83-00233
!topic Dynamic computation of 'MANTISSA for fixed point subtypes

!summary 84-03-16

If F is a non-static fixed point subtype, F'MANTISSA must, in general, be computed at run-time.

!question 84-07-13

The model numbers of a fixed point type depend on the delta and range in a fixed point constraint [3.5.9(6)]. Consider the following declaration:

```
type F is delta 1.0 range -127.0 .. 127.0;
subtype S is F delta 1.0 range L..R;
```

The model numbers of S are determined by the values of L and R, and therefore, S'MANTISSA must, in general, be computed at run time, since L and R need not be static. Is this the intent?

!response 84-01-10

The analysis is correct. S'MANTISSA must, in general, be computed at run-time.

is stated to be equivalent to:

```
type %FP is new %PDF;
subtype F is %FP range %FP(-1.0) .. %FP(1.0);
```

An implementation is allowed to choose the predefined type %PDF such that all and only the model numbers of F are representable, e.g., %PDF might occupy just 16 bits (see AI-00143). In such a case, the upper bound of %FP, the base type of F, is 1.0-2.0**(-15), i.e., the value 1.0 lies just outside the range of the base type. A conversion of a value that lies outside the target type's base type is allowed to raise NUMERIC_ERROR [4.5.7(7)], so it would seem that the conversion, %FP(1.0) is allowed (or required) to raise NUMERIC_ERROR. However, it was intended that declarations like the one given be accepted for a base type that occupies just 16 bits. It was intended to allow programmers to write:

```
type F is delta 2.0**(-15) range -1.0 .. 1.0;
```

rather than requiring programmers to write:

```
type F is delta 2.0**(-15) range -1.0 .. 1.0-2**(-15);
```

In short, the declaration given in the question is legal and neither NUMERIC_ERROR nor any other exception can be raised by such a declaration. (Of course, even if the type declaration is accepted, NUMERIC_ERROR could be raised for F'(1.0).)

Ada Commentary ai-00145-ra.wj downloaded on Thu May 12 16:45:03 EDT 1988

!standard 03.05.09 (14) 86-07-23 AI-00145/04
!class ramification 84-01-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 84-01-10
!references 83-00233
!topic Dynamic computation of 'MANTISSA for fixed point subtypes

!summary 84-03-16

If F is a non-static fixed point subtype, F'MANTISSA must, in general, be computed at run-time.

!question 84-07-13

The model numbers of a fixed point type depend on the delta and range in a fixed point constraint [3.5.9(6)]. Consider the following declaration:

```
type F is delta 1.0 range -127.0 .. 127.0;
subtype S is F delta 1.0 range L..R;
```

The model numbers of S are determined by the values of L and R, and therefore, S'MANTISSA must, in general, be computed at run time, since L and R need not be static. Is this the intent?

!response 84-01-10

The analysis is correct. S'MANTISSA must, in general, be computed at run-time.

Ada Commentary ai-00147-ra.wj downloaded on Fri May 13 09:45:01 EDT 1988

!standard 03.05.09 (18) 87-08-06 AI-00147/05
!class ramification 84-01-10
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-05
!status received 84-01-10
!references 83-00232
!topic Declaring a fixed point type that occupies one word

!summary 86-10-05

A fixed point type that occupies a full word can be declared as:

```
DEL : constant := 1.0/2**(WORD_LENGTH - 1);
type FRACTION is delta DEL range -1.0 .. 1.0 - DEL;
```

!question 86-10-05

3.5.9(18) gives the following declarations as the way to declare a fixed point type that fully occupies one word:

```
DEL : constant := 1.0/2**(WORD_LENGTH - 1);
type FRACTION is delta DEL range -1.0 .. 1.0 - DEL;
```

It does not seem that this type declaration is correct, since the base type must be chosen so -1.0 can be represented as a model number. Shouldn't the declaration be as follows?

```
type FRACTION is delta DEL range -1.0 + DEL .. 1.0 - DEL;
```

!response 87-06-04

The example as written is correct. 3.5.9(6) requires that the mantissa length be chosen

> as the smallest integer number for which each bound of the specified range is either a model number or lies at most SMALL distant from a model number.

For the type FRACTION, SMALL is equal to DEL. The smallest model number for type FRACTION is -1.0 + DEL, which does lie at most SMALL from the lower bound, -1.0, when the mantissa length is WORD_LENGTH - 1. (Note that -1.0 itself is not a model number.)

Ada Commentary ai-00148-ra.wj downloaded on Tue May 10 09:50:02 EDT 1988

!standard 03.06.01 (02) 86-07-23 AI-00148/05
!class ramification 84-02-06
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 84-01-10
!references 83-00211, 83-00216, 83-00218, 83-00219, 83-00220, 83-00228, 83-00309
!topic Legality of -1..10 in loops

!summary 84-07-13

The range -1..10 is illegal as the discrete range in an iteration rule, constrained array type definition, and entry family declaration, since -1 is an expression having a form prohibited by 3.6.1(2), and the other rules of the language do not determine a unique type for the bounds. The possibility of adopting a more liberal rule in a future version of the language will be studied. Note, however, that instead of writing -1..10, one can always write INTEGER range -1..10 or declare -1 as a constant and use the constant name in place of the expression, -1; often it will be appropriate to use the attribute 'RANGE in place of an explicit range such as -1..10.

!question 84-07-13

Is it really intended that -1..10 be an illegal discrete range in an iteration rule, even when INTEGER is the only predefined non-universal integer type?

!response 84-03-16

Yes, this is the intent. Moreover, there is no ambiguity in the wording, there are no implementation difficulties in enforcing the rule, and the validation tests check that such ranges are illegal in this context. Finally, the rule does not restrict programming power since one can always write INTEGER range -1 ..10 or declare -1 as a constant and use the constant name in place of the expression, -1; often it will be appropriate to use the attribute 'RANGE in place of an explicit range such as -1..10.

There is no ambiguity when INTEGER is the only predefined non-universal integer type. The only rules that can be used to resolve the type of the bounds are given explicitly in 3.6.1(2): the type must be discrete and both bounds must have the same type. There are always at least two integer types that satisfy these rules, INTEGER and universal_integer, so -1..10 is illegal even when INTEGER is the only non-universal type that has been declared.

Ada Commentary ai-00149-bi.wj downloaded on Tue May 10 09:50:03 EDT 1988

!standard 03.09 (06) 87-02-23 AI-00149/09
!class binding interpretation 84-01-10
!status approved by WG9/AJPO 87-02-20
!status approved by Director, AJPO 87-02-20
!status approved by Ada Board (20-0-3) 87-02-19
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-05
!status work-item 84-11-06
!status received 84-01-10
!references 83-00221, 83-00222, 83-00224, 83-00311, 83-00425, 83-00530
!topic Activating a task before elaboration of its body

!summary 85-09-21

If an attempt is made to activate a task before its body has been elaborated, PROGRAM_ERROR is raised.

If more than one task is to be activated, the check for unelaborated bodies is performed before an attempt is made to activate any task. Consequently, if PROGRAM_ERROR is raised, no tasks have been activated.

!question 84-11-06

Is the attempt to access the body of a task an action that occurs prior to the task's activation or is it a part of the task's activation? (If considered to be part of a task's activation, then any attempt to activate a task whose body has not been elaborated causes TASKING_ERROR, not PROGRAM_ERROR, to be raised as a result of the activation attempt.)

!recommendation 85-07-30

The attempt to access the body of a task occurs prior to the process of activating the task.

!discussion 85-10-06

3.9(6 and 8) say that PROGRAM_ERROR is raised "for the activation of a task" if the body of the corresponding task unit has not already been elaborated. 9.3(3) says that if an exception is raised "by the activation" of a task, TASKING_ERROR is raised after completion of the activation of any other tasks. It is easy to envision a set of tasks to be activated, some of which raise PROGRAM_ERROR (because their bodies have not been elaborated) and some of which raise, say, NUMERIC_ERROR, when their activation is attempted. For example:

```
task type EXCP;          -- will raise exception during activation

task type PROG_ERR;      -- body will not be elaborated soon enough

type REC is
     record
          A : EXCP;
```

```
            B : PROG_ERR;
        end record;

    task body EXCP is
        NUM : INTEGER := 1/0;
    begin
        null;
    end;

    package P is
        OBJ : REC;
    end P;

    package body P is
    begin
        -- (1) attempt to activate OBJ.A and OBJ.B
    exception
        -- will PROGRAM_ERROR be propagated here?
    end P;
```

If we consider the attempt to access the body of a task to be part of the activation process, then it is clear that 3.9's PROGRAM_ERROR only serves to ensure that TASKING_ERROR will be raised, i.e., at the point indicated by (1), activation of OBJ.A and OBJ.B is attempted in parallel [9.3(1)]. The activation of OBJ.A will raise NUMERIC_ERROR, so 9.3(3) says TASKING_ERROR must be raised at the point of the comment. The attempt to activate OBJ.B will raise PROGRAM_ERROR according to 3.9(6,8). Since PROGRAM_ERROR is raised "by the activation" of a task, this exception is not propagated outside the task unit; 9.3(3) applies and a single TASKING_ERROR exception is raised.

If, on the other hand, we consider the attempt to access a task's body to be an action that occurs prior to activation (since 9.3(1) says that activation consists of elaborating the declarative part, and such an action can only take place after the task body has been accessed), then an implementation must first check to see if all bodies about to be activated have been elaborated. If they have all been elaborated, then PROGRAM_ERROR will not be raised, and activation of the tasks can proceed in parallel; if one or more bodies have not been elaborated, presumably only one PROGRAM_ERROR exception is to be raised. Note that an implementation cannot simply check for an unelaborated body just before activating each task, since then 3.9(6) would require that PROGRAM_ERROR be raised (for unelaborated bodies) and 9.3(3) would require that TASKING_ERROR be raised if other exceptions were raised during the activation of some of the tasks. It was not the intent of the design to have different exceptions raised simultaneously.

A straightforward reading of 3.9(6) implies that PROGRAM_ERROR will be raised when attempting to activate a task whose body has not yet been elaborated. Moreover, the implementation burden of checking for unelaborated bodies is not great, since only one check need be made for each body (not one for each task that is to be activated).

Since tasks are activated in parallel (9.3(1)) and since the check for the elaboration of a task's body occurs prior to the activation of a task, all elaboration checks must be completed before any activation attempt begins. Consequently, if PROGRAM_ERROR is raised, no tasks have been activated.

Ada Commentary ai-00150-bi.wj downloaded on Tue May 10 10:45:01 EDT 1988

!standard 04.08 (05) 86-07-23 AI-00150/04
!class binding interpretation 84-01-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-06-28
!status work-item 84-06-11
!status received 84-01-10
!references 83-00223
!topic Allocated objects belong to the designated subtype

!summary 84-09-10

CONSTRAINT_ERROR is raised if an object specified by an allocator does not belong to the designated subtype for the allocator.

!question 84-09-10

Consider the following example:

```
type A is access STRING (1..10);
X : A := new STRING (1..4);     -- CONSTRAINT_ERROR?
```

4.8(5) says "If the allocator includes a subtype indication, the created object is constrained either by the subtype or by the default discriminant values." Is it intended that no CONSTRAINT_ERROR be raised even though the object specified by the allocator has index constraint 1..4 (instead of 1..10 as specified in A's type definition)? Moreover, is it the intent that

```
X := new STRING(2..11);
```

raise CONSTRAINT_ERROR?

!recommendation 84-09-10

If the designated type for an access type is a constrained array type or a constrained type with discriminants and an allocator for the access type includes a subtype indication, a check is made that the index bounds or discriminant values imposed by the subtype indication are the same as those specified for the access type's designated subtype. CONSTRAINT_ERROR is raised if this check fails. (It is not defined whether this check is performed before or after creation of a designated object.)

!discussion 84-09-10

It is the intent of the Standard that objects designated by access values always satisfy any subtype constraint imposed on the access type. Since new STRING(1..4) implies creation of an object with bounds 1..4, and since A

requires all designated objects to have bounds 1..10, X := new STRING(1..4) should raise CONSTRAINT_ERROR, as should new STRING (2..11) in the assignment:

```
X := new STRING (2..11);
```

The same reasoning applies to types with discriminants.

Note that even if the subtype indication in an allocator does not impose a discriminant constraint, CONSTRAINT_ERROR must sometimes be raised:

```
type DEF (D : INTEGER := 0) is ... end;
type A_CDEF is access DEF(1);
Y : A_CDEF;
...
Y := new DEF;    -- CONSTRAINT_ERROR raised.
```

CONSTRAINT_ERROR must be raised because the default discriminant value for DEF does not satisfy the constraint imposed by the declaration of A_CDEF.

Ada Commentary ai-00151-bi.wj downloaded on Tue May 10 10:45:02 EDT 1988

!standard 05.04 (03) 86-07-23 AI-00151/05
!class binding interpretation 84-01-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 84-01-10
!references 83-00227, 83-00310
!topic Case expression of a type derived from a generic formal type

!summary 84-03-16

A type derived from a generic formal type cannot be used in the expression of a case statement.

!question 84-07-13

The Standard forbids an expression of a generic formal type in the expression of a case statement, but does not forbid expressions of a type derived from a generic formal type. Is this an oversight?

!recommendation 84-03-16

The intention was to forbid case expressions of a type derived from a generic formal type.

!discussion 84-03-16

The Standard says that the expression of a case statement "must not be of a generic formal type." It further states in 12.1(3) that the term "generic formal type" is used to refer to corresponding generic formal parameter, and hence, this rule does not include types derived, directly or indirectly, from a generic formal type.

The reason for forbidding expressions of a generic formal type in the expression of a case statement is that each value of the expression's type must be represented once and only once in the set of choices, and no other value is allowed. This check cannot be determined at compile-time for a generic formal parameter, since the range of values covered by such a parameter's base type can vary from one instantiation to the next. Hence, case expressions were not allowed to be of a generic formal type. The same reasoning applies to types derived from a generic formal type; failure to mention such types was an oversight.

Ada Commentary ai-00153-bi.wj downloaded on Tue May 10 10:50:01 EDT 1988

!standard 07.04.01 (04) 86-07-23 AI-00153/05
!class binding interpretation 84-01-10
!status apprcved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status work-item 84-01-29
!references 83-00236, 83-00241
!topic Membership tests cannot use an incompletely declared private type

!summary 84-03-16

The type mark for a private type cannot be used in a membership test before the end of the full declaration of the type. This restriction also applies to the use of a name that denotes a subtype of the private type and the use of a name that denotes any type or subtype that has a subcomponent of the private type.

!question 84-07-13

The wording of this paragraph forbids the use of an incomplete private type within a simple expression. A membership test is an expression, but not a simple expression. Did the Standard mean to say "expression" instead of "simple expression"?

!recommendation 84-03-16

The intent was to exclude use of such types in membership tests (i.e., within an expression as well as within simple expressions).

!discussion 84-03-16

Consider the following example:

```
package P is
     type T (D : INTEGER) is private;
     subtype ST is T(-1);
     C, D : constant T;
     procedure P (X : BOOLEAN := C in ST);    -- legal?
     procedure Q (X : BOOLEAN := (C in ST)); -- illegal
```

The parenthesized expression in Q is illegal, since a parenthesized relation is syntactically a simple expression, and so is forbidden by the current wording; the unparenthesized form is not a simple expression according to the syntax in 4.4.

The intent was to forbid use of type marks such as T and ST in expressions as well as simple expressions; allowing such usage just within unparenthesized membership tests was an oversight.

Ada Commentary ai-00154-ra.wj downloaded on Tue May 10 10:50:02 EDT 1988

!standard 07.04.02 (08) 86-07-23 AI-00154/06
!standard 07.04.02 (07)
!class ramification 84-01-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-05-13
!status committee-approved (8-0-1) 85-02-27
!status work-item 85-02-05
!status received 84-01-10
!references AI-00139, AI-00260, 83-00231, 83-00476, 83-00479, 83-00487
!topic Additional operations for composite and access types

!summary 85-03-04

This Commentary gives details showing which operations are declared for certain composite and access types immediately after their declaration and which are declared later in a package body.

!question 85-05-27

Consider the following example:

```
package P is
     type T is private;

     package Q is
          type ARR1 is array (1..10) of T;
          type ARR2 is array (1..10, 1..10) of T;
          type REC (D : INTEGER) is
               record
                    C : T;
               end record;
          type ACC_AR1 is access ARR1;
          type ACC_REC is access REC;
          type ACC_T is access T;
     end Q;
private
     type T is ...;
end P;
```

(1) What operations are declared for each of the composite and access types declared within package Q and what "additional operations" (in terms of 7.4.2(7, 8)) are declared for each type within Q's body (i.e., after T's full declaration)?

(2) 7.4.2(7) gives rules specifying when certain "additional operations" are declared for composite types having a component of a private type if the composite type is declared prior to the full declaration of the private type. 7.4.2(8) says "The same rules apply to the operations that are implicitly declared for an access type whose designated type is a private type or a type

declared by an incomplete type declaration." Does "same rules" refer to the rules given in 7.4.2(6, 7) defining the operations declared for the access type at its point of declaration as well as the "additional operations" declared later after the private type's full declaration?

(3) Are any "additional operations" declared for an access type such as ACC_AR1 and ACC_REC, i.e., an access type whose designated type is a composite type containing a component of a not yet fully declared private type?

!response 85-02-05

(1) The operations implicitly declared for an array type are described in 3.6.2. The wording in 3.6.2 shows which operations depend on characteristics of the component type, and which characteristics are relevant. For example, 3.6.2(1) says the formation of string literals is an operation declared for one-dimensional arrays if the component type is a character type. A private type is clearly not a character type, so string literal formation is not an operation declared for ARR1. If the full declaration of T declares a character type, then the operation involved in string literals will be implicitly declared in the package body for Q. Using this style of reasoning, we can determine which operations are declared in Q's specification and which in the body after T's full declaration:

	ARR1	ARR2
assignment	*	*
aggregates	*	*
membership	spec	spec
indexing	spec	spec
qualification	spec	spec
conversion	spec	spec
slices	spec	no
string literals	body	no
attributes	spec	spec
equality	*	*
catenation	*	no
relational ops	body	no
logical ops/NOT	body	no

key:

* not declared in Q's specification if T is limited private; if not declared in Q's specification, then declared in Q's body if and only if the full declaration of T is not limited.

spec operation is declared in Q's specification

no operation is not declared in Q's specification or body

body operation is declared in Q's body if T's full declaration is suitable, e.g., the logical operators and NOT are declared in Q's body for ARR1 if T is declared as a boolean type.

Operations declared for records are specified in 3.7.4. Using the same key as above, the following operations are declared for the type REC:

assignment	*
aggregates	*
membership	spec
selection (.D, .C	spec
qualification	spec
conversion	spec
attributes	spec
equality	*

Operations declared for access types are specified in 3.8.2:

designated type:	ARR1	ARR2	REC	T
assignment	spec	spec	spec	spec
allocator	spec	spec	spec	spec
membership	spec	spec	spec	spec
qualification	spec	spec	spec	spec
conversion	spec	spec	spec	spec
NULL	spec	spec	spec	spec
.all	spec	spec	spec	spec
discriminant selection	no	no	spec	no
component selection	no	no	spec	body
indexing	spec	spec	no	body
slices	spec	no	no	body
entry selection	no	no	no	body*
array attributes	spec	spec	no	no
task attributes	no	no	no	body*
representation attr.	spec	spec	spec	spec
equality	spec	spec	spec	spec

body* operation is declared if T had been declared as a limited private type whose full declaration was a task type.

Note that if T is declared to be a record type with component C and V is a variable having type ACC_T, then V.all is an operation declared within package Q. The operation for selecting component C directly from V is not declared until package body Q. Hence, outside of package body Q, V.C is illegal (see also AI-00139).

(2) When 7.4.2(8) refers to "the same rules," it is referring to the rules given in the preceding paragraph, namely, the rules defining what additional operations are declared after the full declaration of the designated type.

(3) There is no need for 7.4.2(8) to mention composite designated types that contain a subcomponent of a not yet fully declared private type, since there are no additional operations that can be declared for the corresponding access types (see the table when the designated type is ARR1, ARR2, or REC).

Ada Commentary ai-00155-bi.wj downloaded on Tue May 10 11:45:01 EDT 1988

!standard 03.02.01 (18) 86-12-01 AI-00155/08
!standard 07.04.03 (04)
!class binding interpretation 84-01-10
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status committee-approved 86-06-02 (reviewed)
!status committee-approved 85-12-11 (subject to further editorial review)
!status committee-approved (10-0-0) 85-09-05 (subject to editorial review)
!status work-item 85-02-03
!status received 84-01-10
!references 83-00230, 83-00653
!topic Evaluation of an attribute prefix having an undefined value

!summary 86-05-05

The execution of a program is erroneous if the name of a deferred constant is evaluated before the full declaration of the constant has been elaborated.

Evaluation of the name of a scalar variable is not erroneous, even if the variable has an undefined value, if the name occurs as the prefix for the attribute ADDRESS, FIRST_BIT, LAST_BIT, POSITION, or SIZE. (In these cases, the value of the variable is not needed.)

!question 85-10-31

7.4.3(4) says "The execution of a program is erroneous if it attempts to use the value of a deferred constant before the elaboration of the corresponding full declaration." Is the declaration of V in the example below erroneous?

```
package P is
     type T is private;
     DC : constant T;                      -- deferred constant
     type REC is
          record
               C : INTEGER := DC'SIZE;     -- legal (7.4.3(2))
          end record;
     V : REC;                              -- erroneous?
private
     ...
end P;
```

Note that the value of DC is not needed to determine DC'SIZE, and moreover, 4.1(9) says the evaluation of the prefix DC only determines the entity denoted by the name (not its value).

Now consider:

```
X : INTEGER;
Y : INTEGER := X'SIZE;
```

X has an undefined value. Is the evaluation of X'SIZE erroneous? Note that

3.2.1(18) says: "The execution of a program is erroneous if it attempts to evaluate a scalar variable with an undefined value," and 4.1.4(5) requires "evaluation" of X.

!recommendation 86-05-05

If the prefix of an attribute denotes a constant declared by a deferred constant declaration and evaluation of the prefix is attempted before the deferred constant's full declaration has been elaborated, execution of the program is erroneous.

Evaluation of the name of a scalar variable is not erroneous, even if the variable has an undefined value, if the name occurs as the prefix of an attribute and the value of the variable is not required to determine the attribute's value.

!discussion 86-05-05

First, the declaration of REC is legal according to the wording of 7.4.3(2), which allows the use of a name that denotes a deferred constant "in the default expression for a record component". "In the default expression" includes use as a prefix and as the argument of a function (e.g., the default expression could be F(DC) if F were suitably declared; of course, an attempt to invoke F before elaboration of its body would raise PROGRAM_ERROR). (Note: DC'SIZE cannot be replaced with T'SIZE in the question because 7.4.1(4) forbids the use of an incompletely declared type name "within" a simple expression.)

Second, 13.7.2(5) requires that the prefix of the SIZE attribute denote an object (or a subtype). Elaboration of the declaration of V includes evaluation of the default expression DC'SIZE, but DC at that point does not yet denote an object; only the full declaration of the constant creates an object [3.2.1(7)]. The evaluation of such a prefix should therefore be considered an error. Since this error cannot in general be detected prior to execution, and since it is not reasonable to require that an exception be raised when such a prefix is evaluated, it is best to consider it the programmer's responsibility to avoid having such prefixes evaluated, i.e., evaluation of such prefixes should be considered to make execution of the program erroneous.

Execution of a program is not to be considered erroneous if the prefix of a predefined attribute is a scalar variable and the prefix has an undefined value when it is evaluated. (Such a prefix is only legal if it is the prefix of the attribute ADDRESS, FIRST_BIT, LAST_BIT, POSITION, or SIZE; such an evaluation only determines the entity denoted by the name [4.1(9, 10)] and does not require any value for the variable.) Evaluation of the initialization expression for Y (i.e., evaluation of X'SIZE) was not intended to be considered erroneous even though the prefix has an undefined value.

Ada Commentary ai-00157-bi.wj downloaded on Tue May 10 11:45:03 EDT 1988

!standard 08.07 (07) 86-07-23 AI-00157/05
!class binding interpretation 84-01-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (10-0-0) 85-09-05
!status work-item 85-02-05
!status received 84-01-10
!references AI-00015, 83-00212, 83-00213, 83-00214, 83-00215
!topic Overloading resolution and parenthesized expressions

!summary 85-10-31

The rule requiring aggregates to be given in named notation if they contain a single component association (4.3(4)) is to be considered a syntax rule for purposes of overloading resolution, and in particular, can be used to help resolve the type of a parenthesized expression.

!question 85-12-23

8.7(7) says syntax rules are used in resolving the interpretation of overloaded constructs, but it is not entirely clear whether by "syntax rules" 8.7 means just the context-free rules summarized in Appendix E, or these rules augmented with certain narrative restrictions as described in 1.5(1). For example, consider:

```
type ARR is array (1..2) of INTEGER;
procedure P (X : INTEGER);
procedure P (X : ARR);
...
P((1,2));        -- unambiguous  (1)
P((9));          -- unambiguous? (2)
```

The first case is unambiguous because (1,2) is syntactically an aggregate, and hence has a composite type (as opposed to a scalar type). Knowing (1,2) is potentially of type ARR and not of type INTEGER suffices to resolve P.

In case 2, if we just use the context-free syntax rules of the Standard, P's argument can be parsed as either an aggregate or as a parenthesized expression. Hence, the type of (9) can be either ARR or INTEGER, and the call cannot be resolved. A narrative rule [4.3(4)], however, requires that aggregates containing a single component be given in named notation. If this narrative rule is considered part of the "syntax rules," then the syntactic ambiguity in example (2) is resolved; (9) cannot have type ARR since it is not an aggregate. Consequently, P is unambiguous.

Is (2) ambiguous or not?

!recommendation 85-10-31

Among the syntax rules used in resolving overloaded constructs is the rule given in 4.3(4) that requires aggregates containing a single component association to be given in named notation.

!discussion 85-10-31

During the final revision of the language, comment #4979 explicitly mentioned that the rule regarding the form of one-component aggregates should be used in resolving subprogram and entry calls. The response to the comment said that the rules given in the draft Standard were believed to cover the cases mentioned in the comment.

There can be little doubt that the intent of the rule given in 4.3(4) was to make examples like that shown in (2) unambiguous, i.e., the intent was to interpret the rule in 4.3(4) as a syntax rule, at least for purposes of overloading resolution.

Ada Commentary ai-00163-ra.wj downloaded on Tue May 10 11:50:01 EDT 1988

!standard 04.09 (06) 86-07-23 AI-00163/05
!class ramification 84-01-13
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-20
!status work-item 86-01-23
!status received 84-01-13
!references 83-00244, 83-00244, 83-00283
!topic Implicit conversion preserves staticness

!summary 86-03-05

A static expression is static even if an implicit conversion is applied to it.

!question 86-01-28

Consider the following example:

```
C1 : constant NATURAL := 1;              -- (1)
C2 : constant NATURAL := INTEGER (1);    -- (2)
```

According to 4.6(15) case (1) implies a conversion of the literal appearing on the right hand side of the assignment. Thus, case (1) is semantically equivalent to (2). C2 is not allowed in a static expression because INTEGER(1) is not a static expression. Since the initialization of C1 also involves a conversion, is C1 allowed in a static expression?

!response 86-01-28

4.9(2-10) give the criteria defining which expressions are static. These criteria are specified in syntactic terms. In (1) the constant C1 is explicitly declared by a constant declaration with a static subtype. To include the constant C1 in a static expression, it must satisfy the syntactic criteria specified in 4.9(2-10). In particular, it must adhere specifically to rule (d) which requires that the initialization expression be a static expression. In (1) the expression used for the initialization is a static expression since there are no operators (implicit conversion is an operation) and the only primary in the expression is a numeric literal, which is a valid primary in a static expression.

Explicit conversion is not allowed in a static expression since type conversion is not one of the primaries listed as a permitted primary by the rules (a) through (f) in 4.9. Since implicit conversion does not change the syntax of an expression, implicit conversion can be applied to static expressions without affecting their status.

Ada Commentary ai-00167-co.wj downloaded on Tue May 10 11:50:03 EDT 1988

!standard 09.04 (00) 86-07-23 AI-00167/04
!class confirmation 84-03-16
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status board-approved 84-06-29
!status committee-approved 84-02-06
!status received 84-01-13
!references 83-00247
!topic It is possible to access a task from outside its master

!summary 84-03-16

A task can be accessed from outside its master, since a function can return a task object as its value, even a task that was activated inside the function.

!question 84-07-13

A task object is allowed as the returned value of a function. Must the following program print "terminated"?

```
with Text_IO; use Text_IO;
procedure Main is
   task type T;
   task body T is
   begin
      null;
   end T;

   function F return T is
      X : T;
   begin
      return X;
   end F;

begin
   if F'Terminated then
      Put_Line("terminated");
   else
      Put_Line("not terminated");
   end if;
end;
```

!response 84-03-16

The above example is correct; the program should print "terminated". However, to avoid retaining space for such tasks, an implementation could keep a "terminated" flag in the descriptor for the task, i.e., an implementation could treat X as a record: one component would point to the task body and another component would indicate whether the designated task is

terminated. Thus any space for the task object designated by X could be released when F is exited.

Ada Commentary ai-00169-ra.wj downloaded on Tue May 10 12:45:02 EDT 1988

!standard 04.03 (06) 86-07-23 AI-00169/06
!class ramification 85-09-04
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (7-0-2) 85-09-04
!status work-item 84-06-11
!status received 84-01-17
!references 83-00254, 83-00255
!topic Legality of incomplete null multidimensional array aggregates

!summary 85-09-20

A multidimensional aggregate is illegal if a value is omitted in the specification of any dimension.

!question 85-09-20

4.3(6) says "each component of the value defined by an aggregate must be represented once and only once in the aggregate." A null array has no components, so is the following aggregate legal or not when its type is a multidimensional array type?

```
(1..2 => (2..1 => 1),
 -- 3 is deliberately omitted
 4..5 => (2..1 => 2))
```

!response 85-10-17

The syntactic term, aggregate, applies only to one dimensional aggregates, in accordance with the usual rule that use of a syntactic term in the text refers to the syntactic construct [1.5(6)]. Therefore, the rule stated in 4.3(6) (that "each aggregate must be complete") applies to aggregates in their one-dimensional form. Hence,

```
(1..2 => (2..1 => 1),
 4..5 => (2..1 => 2))
```

is illegal regardless of the nature of the expressions specified in the outer component associations, and in particular, regardless of whether or not the inner aggregates are subaggregates of a null multidimensional array.

Ada Commentary ai-00170-bi.wj downloaded on Tue May 10 12:45:03 EDT 1988

!standard 08.05 (05) 86-11-19 AI-00170/06
!class binding interpretation 84-01-17
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-20
!status work-item 86-01-17
!status received 84-01-17
!references 83-00257
!topic Renaming a slice

!summary 86-01-17

A slice must not be renamed if renaming is prohibited for any of its components.

!question 86-03-05

4.1.2(1) says that:

> A slice denotes a one-dimensional array formed by a sequence of consecutive components of a one-dimensional array.

and 8.5(5) says that:

> The following restrictions apply to the renaming of a subcomponent that depends on discriminants of a variable. ...

A strict reading of 4.1.2 indicates that a slice is not an example of a component -- it is, instead, a one-dimensional array, which is apparently an entirely separate concept. If so, then apparently the restrictions in 8.5(5) about renaming a subcomponent that depends on a discriminant do not apply to slices. For example:

```
type SINT is range 0..100;
type VREC (N : SINT := 0) is
     record
          S : STRING (1..N);
     end record;

OBJ : VREC := (3, "ABC");

OBJ1 : CHARACTER renames OBJ.S(1);       -- illegal by 8.5(5)
OBJ2 : STRING    renames OBJ.S(1..2);    -- illegal? (yes)
```

OBJ.S(1..2) is a slice and thus not clearly a "component", so 8.5(5) does not clearly apply. Do the restrictions of 8.5(5) apply to slices as well as "just subcomponents"?

!recommendation 86-01-17

A slice must not be renamed if renaming is prohibited for any of its components.

!discussion 86-03-05

The reason for the restrictions in 8.5(5) is to prevent the newly declared name from denoting an object whose existence may subsequently cease while execution is still within the scope of the name. In the example, a subsequent assignment

```
OBJ := (0, "");
```

would cause OBJ1 to denote a no-longer existing object, namely, OBJ.S(1). Similarly, OBJ2 would now denote a non-existent array object, namely, OBJ.S(1..2). This undesirable situation was not intended.

!appendix 86-11-19

****

!section 08.05 (05) Geoff Mendal/Standford 86-11-07 83-00859
!version 1983
!topic Error in AI-00170/05

The example uses the bound N, which is of the type SINT.
Bounds on the type STRING must be of the type INTEGER.
Perhaps the intent was to make the type SINT a subtype
of POSITIVE.

gom

- - - - End forwarded message - - - -

Ada Commentary ai-00172-ra.wj downloaded on Tue May 10 12:50:02 EDT 1988

!standard 14.03.06 (13) 86-07-23 AI-00172/06
!class ramification 84-11-28
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-11-06
!status received 84-01-17
!references 83-00258, 83-00263, 83-00384, 83-00396, 83-00464, 83-00465, 83-00489, 83-00484
!topic GET_LINE for interactive devices

!summary 84-12-31

An implementation is allowed to assume that certain external files do not contain page terminators. Such external files might be used to represent interactive input devices. (To ensure that such files have no page terminators, an implementation may refuse to recognize any input sequence as a page terminator.) Under such an assumption, GET_LINE and SKIP_LINE can return as soon as a line terminator is read.

!question 84-12-31

When reading input from an interactive device using GET_LINE or SKIP_LINE, must an implementation always wait after a line terminator is read to see if a page terminator will also be input?

!response 85-05-27

The specification of GET_LINE states that it reads characters until "the end of the line is met, in which case the procedure SKIP_LINE is then called (in effect) with a spacing of one". The specification of SKIP_LINE includes a check for whether the end of line is followed by an end of page, in which case the page is to be skipped as well as the line. Since SKIP_LINE is in effect called by GET_LINE, GET_LINE also must skip over any page terminator that follows a line terminator.

When GET_LINE is called for a file that is associated with an interactive terminal, the possibility of having to skip a page terminator appears to mean that before GET_LINE can return, the terminal user must enter additional input after inputting a line terminator. If input is line buffered, a user must enter a complete second line before the first line can be processed by the Ada program.

One might think this problem could be solved by "lazy lookahead," i.e., by returning from GET_LINE before checking to see what follows the line terminator and then waiting at the next read operation to see if a page terminator should be skipped. This implementation would not be correct, since a call to LINE or PAGE will return the wrong value while the lookahead is still pending (e.g., LINE should return the value 1 if the line terminator

accepted by GET_LINE is followed by a page terminator). To be correct, if a lookahead is pending, LINE and PAGE must wait until the lookahead can be resolved. Thus an operation that does not seemingly need to read from the terminal (i.e., LINE and PAGE) would actually delay a program until it can be determined whether the line terminator is followed by a page terminator. Such behavior would be considered surprising by many users.

Another seeming alternative is to use a single ASCII control character (e.g., <FF>) to represent a line terminator followed by a page terminator. Suppose a line terminator is represented by the single ASCII control character, <CR>. Then GET_LINE can return immediately after it reads either a <CR> or <FF>. <CR> immediately followed by <FF> represents two line terminators followed by a page terminator. The problem with this alternative is insuring the same conventions are followed when outputting text. When NEW_LINE is called, no characters can be output until it is known whether the next call will be to NEW_PAGE. The sequence of calls, NEW_LINE, NEW_PAGE, must output the single control character, <FF>, but NEW_LINE followed by PUT must output just <CR>. In short, the delays now show up for interactive output rather than interactive input.

An alternative that has been considered is to interpret the semantics of NEW_PAGE so NEW_PAGE always outputs a line terminator followed by a page terminator, even if a NEW_PAGE call is immediately preceded by a call to NEW_LINE. This interpretation would allow NEW_LINE to always output <CR> and NEW_PAGE to always output <FF>. Such behavior would not, however, be consistent with the specification in 14.3.4(15), which says (for NEW_PAGE): "Outputs a line terminator if the current line is not terminated or if the current page is empty. Then outputs a page terminator."

The recomended solution is to assume page terminators do not exist in the input file. Such an assumption might be made just for files that are associated with interactive terminals. Under this assumption, GET_LINE could return immediately after reading a line terminator, since a page terminator could never be input. The disadvantage of this solution is that GET_LINE must know whether it is being invoked for an "interactive" file. In modern systems, with logical files, virtual terminals, pipes, and networks, it is often impossible to know what ultimate device is associated with an external file. Of course, an implementation could assume that STANDARD_INPUT was "interactive", and otherwise use the FORM argument when OPENing a text file for input to say whether the existence of page terminators should be assumed or not.

(The problem would not exist for GET_LINE if GET_LINE called SKIP_LINE before reading characters. This actually was the specification for GET_LINE in the July 1982 Draft of the Ada Standard. The specification was changed to its current form in response to comment #5206, which pointed out that with the 1982 semantics, GET_LINE would skip over an empty line rather than return a value in LAST indicating that the line was empty.)

Ada Commentary ai-00173-bi.wj downloaded on Tue May 10 12:50:04 EDT 1988

!standard 09.04 (05) 86-12-01 AI-00173/05
!class binding interpretation 84-01-17
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (9-0-1) 86-05-13 (pending editorial review)
!status work-item 86-01-20
!status received 84-01-17
!references 83-00001, 83-00120, 83-00851
!topic Completion of execution by exception propagation

!summary 86-07-01

The execution of a task, block statement, or subprogram is completed if an exception is raised by the elaboration of its declarative part.

The execution of a task, block statement, or subprogram is completed if an exception is raised before the first statement following the declarative part and there is no corresponding handler, or if there is one, when it has finished the execution of the corresponding handler. (TASKING_ERROR is the only exception that can be raised under these circumstances. It can be raised by unsuccessful attempts to activate one or more dependent tasks after elaboration of the declarative part and before beginning execution of the sequence of statements.)

!question 86-07-01

9.4(5) defines the conditions under which a task, block, or subprogram completes its execution. The paragraph does not seem to cover all the relevant conditions. In particular, if an exception is raised in a declarative part of a task, block, or subprogram, completion does not seem to be defined. Similarly, completion seems to be undefined if an exception is raised after the elaboration of the declarative part and before execution of the sequence of statements has begun. (An exception can be raised under these circumstances by the attempt to activate tasks declared in the declarative part.)

It is important to define completion in these cases since execution cannot leave a completed block or subprogram until all dependent tasks are terminated. Similarly, a task cannot terminate until it becomes completed (and all dependent tasks are terminated). If the definition of completion is not extended to cover these cases, it is unclear what effect is required by the Standard when an exception is raised under these conditions.

Are the definitions given in 9.4(5) incomplete? If so, what are the intended definitions?

!recommendation 86-08-05

The execution of a task, block statement, or subprogram is completed if an exception is raised by the elaboration of its declarative part.

If an exception is raised after elaborating the declarative part of a task, block statement, or subprogram, and before executing the first statement following the declarative part, then the execution of the task, block statement, or subprogram is completed if there is no corresponding handler; if there is a handler, execution is completed when the task, block, or subprogram has finished executing the handler.

!discussion 86-01-20

The wording of 9.4(5) does not cover all the necessary cases. A task, block, or subprogram should be considered to have completed its execution if an exception is raised while elaborating the declarative part. If such an exception is raised, the block or subprogram cannot be left until all of its dependent tasks are terminated. For example:

```
declare
     type ACC_TSK is access TSK_TYPE;
     ORPHAN_TASK : ACC_TSK := new TSK_TYPE; -- activate task
     OBJ         : NATURAL := -1;           -- raise exception
begin
     null;
end;          -- wait for ORPHAN_TASK to terminate
```

Similarly, if a task raises an exception during its activation, it cannot be terminated until all its activated dependent tasks (if any) are terminated (as is assumed by 9.3(3)):

```
task TSK;

task body TSK is
     type ACC_TSK is access TSK_TYPE;
     ORPHAN_TASK : ACC_TSK := new TSK_TYPE; -- activate task
     OBJ         : NATURAL := -1;           -- raise exception
begin
     null;
end;          -- wait for ORPHAN_TASK to terminate
```

Even though TSK cannot be successfully activated, it cannot be terminated until ORPHAN_TASK terminates.

A similar omission in wording occurs when tasks are activated just before execution of the sequence of statements of a task, block, or subprogram. The last sentence of 9.4(5) says:

> Finally the execution of a task, block statement, or subprogram is completed if an exception is raised by the execution of its sequence of statements and there is no corresponding handler, or, if there is one, when it has finished the execution of the corresponding handler.

This wording does not cover the possibility that TASKING_ERROR can be raised just before execution of the sequence of statements begins. For example:

```
declare
     TSK_OBJ_1 : SOME_TASK_TYPE;
     TSK_OBJ_2 : ANOTHER_TASK_TYPE;
begin
     PUT_LINE ("I am executing");
end;
```

If TSK_OBJ_1 is successfully activated but an exception is raised during the activation of TSK_OBJ_2, TASKING_ERROR will be raised before PUT_LINE is called, i.e., before execution of the block's sequence of statements has started. The intention in this case is that the exception be considered to complete the block's execution. Since the block is completed and since TSK_OBJ_1 was activated successfully, the block will be left only after TSK_OBJ_1 is terminated.

Ada Commentary ai-00177-bi.wj downloaded on Tue May 10 13:45:03 EDT 1988

!standard 04.03.02 (08) 86-07-23 AI-00177/04
!class binding interpretation 84-01-19
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-06-28
!status work-item 84-06-11
!status received 84-01-19
!references 83-00261
!topic Use of others in a multidimensional aggregate

!summary 84-12-10

An others choice is allowed if an aggregate is not a subaggregate and is the expression of a component association of an enclosing (array or record) aggregate. An others choice is also allowed if an aggregate is a subaggregate of a multidimensional array aggregate that is in one of the contexts specified by 4.3.2(5-8).

!question 84-09-10

Consider the following declarations:

```
type A_2dim is array (INTEGER range <>, INTEGER range <>)
                                        of STRING (1..4);
type A_1dim is array (INTEGER range <>) of STRING (1..4);
procedure P (Var : A_2dim);
procedure Q (Var : A_1dim);
```

Which of the following aggregates are illegal because of their use of OTHERS?

```
P (Var => (1 => (2 => (others => 'a') )) );
Q (Var => (1 => (others => 'a')));
P (Var => (1 => (others => "abcd")));
```

!recommendation 84-12-10

An others choice is allowed if an aggregate is not a subaggregate and is the expression of a component association of an enclosing (array or record) aggregate. An others choice is also allowed if an aggregate is a subaggregate of a multidimensional array aggregate that is in one of the contexts specified by 4.3.2(5-8). Consequently, the following aggregates are legal because (others => 'a') is not a subaggregate:

```
P (Var => (1 => (2 => (others => 'a') )) );     -- legal
Q (Var => (1 => (others => 'a')));      -- legal
```

but the aggregate in the next call is illegal because (others => "abcd") is a

subaggregate of a multidimensional array aggregate and P's formal parameter is not constrained, so the multidimensional aggregate does not appear in one of the permitted contexts:

```
P (Var => (1 => (others => "abcd")));   -- illegal
```

!discussion 84-09-10

4.3.2(8) specifies one of the conditions under which an others choice is allowed in an aggregate, namely, an others choice is allowed if:

> "The aggregate is the expression of the component association of an enclosing (array or record) aggregate. Moreover, if this enclosing aggregate is a multidimensional array aggregate then it is itself in one of these three contexts."

The aggregate enclosing (others => 'a') is certainly a multidimensional array aggregate in the call to P, and this multidimensional aggregate does not appear in one of the three contexts permitted by 4.3.2(5-8) so 4.3.2(8) seems to say that the first call to P contains an illegal others choice. But the aggregate in the call to Q is legal because the (others => 'a') aggregate appears as the component association of an enclosing ONE-dimensional array aggregate.

When (others => 'a') is not a subaggregate of a multidimensional array, it is inconsistent to allow its occurrence only to specify component values for one-dimensional arrays. The intent was to allow such an aggregate except when it is being used as a subaggregate of a multidimensional array aggregate, e.g.,

```
P (Var => (1 => (others => "abcd")) );   -- illegal
```

The rule in 4.3.2(8) should have distinguished whether the component association occurs in a subaggregate or not. In particular, only if an aggregate is a subaggregate does the legality of a component association specifying an others choice depend on the context in which the containing multidimensional aggregate appears.

Ada Commentary ai-00179-ra.wj downloaded on Tue May 10 13:45:05 EDT 1988

!standard 03.05.10 (08) 86-10-02 AI-00179/07
!class ramification 85-09-05
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-05
!status work-item 84-06-11
!status received 84-01-25
!references 83-00265, 83-00654
!topic The definition of the attribute FORE

!summary 85-09-18

The attribute 'FORE is defined in terms of the decimal representation of model numbers.

!question 85-09-18

For a fixed point type definition such as

```
type F is delta 0.1 range 0.0 .. 9.96;
for F'SMALL use 0.01;
```

is the value of F'FORE 2 or 3? Note that when outputting F'LAST with an AFT of 1, the string " 10.0" will be produced, and this string requires a FORE of 3.

!response 85-10-02

The Standard gives the following definition for 'FORE:

> Yields the minimum number of characters needed for the integer part of the decimal representation of any value of the subtype T, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. ...

For a fixed point subtype declared as follows:

```
type F is delta 0.1 range 0.0 .. 9.96;
for F'SMALL use 0.01;
```

the value of 'FORE is 2 since the straightforward interpretation of "decimal representation of any value of the subtype" means the exact decimal representation of the model numbers belonging to F. In this case, the value 2 is unsuitable when certain values, e.g., 9.96, are output with an AFT of 1. It is up to the programmer to take this effect into account when using fixed point output formats.

The value returned by 'FORE can be implementation dependent. For example:

```
type G is delta 0.01 range 1.00 .. 10.00;
for G'SMALL use 0.01;
subtype SG is F delta 0.1 range 1.0 .. 9.95;
```

For the subtype SG, 9.9 and 10.0 are consecutive model numbers (3.5.9(14)). It is implementation dependent whether the upper bound of SG is represented as the model number 9.9 or the model number 10.0. Depending on the implementation's choice, the value returned by SG'FORE will be either 2 or 3. In addition, note that the bounds of SG need not be given by static expressions. If the upper bound is non-static and has a value lying in the model interval 9.9 to 10.0, SG'FORE's value will be implementation dependent (and must be computed at run-time). The fact that 'FORE may return implementation dependent values should be taken into consideration by programmers.

!appendix 86-10-02

****

!section 03.05.10 (08) J. Goodenough 86-09-07 83-00803
!version 1983
!topic Correction to AI-00179/06
!reference AI-00179/06

The discussion section of this AI contains an error. The following example is given:

```
type G is delta 0.01 range 1.00 .. 10.00;
for G'SMALL use 0.01;
subtype SG is F delta 0.1 range 1.0 .. 9.95;
```

The discussion says:

> For the subtype SG, 9.9 and 10.0 are consecutive model numbers (3.5.9(14)). It is implementation dependent whether the upper bound of SG is represented as the model number 9.9 or the model number 10.0. Depending on the implementation's choice, the value returned by SG'FORE will be either 2 or 3. In addition, note that the bounds of SG need not be given by static expressions. If the upper bound is non-static and has a value lying in the model interval 9.9 to 10.0, SG'FORE's value will be implementation dependent (and must be computed at run-time). The fact that 'FORE may return implementation dependent values should be taken into consideration by programmers.

The discussion is incorrect; the model numbers for SG are the same as the model numbers for type G because SMALL is explicitly specified with a length clause. For the discussion to be correct, the example should be changed so the upper bound of SG is 9.995. The model interval in the discussion will then range from 9.99 to 10.00. After these corrections are made, the example and discussion should read as follows:

```
type G is delta 0.01 ragne 1.00 .. 10.00;
for G'SMALL use 0.01;
subtype SG is F delta 0.01 range 1.00 .. 9.995;
```

For the subtype SG, 9.99 and 10.00 are consecutive model numbers (3.5.9(14)). It is implementation dependent whether the upper bound of SG is represented as the model number 9.99 or the model number 10.0. Depending on the implementation's choice, the value returned by SG'FORE will be either 2 or 3. In addition, note that the bounds of SG need not be given by static expressions. If the upper bound is non-static and has a value lying in the model interval 9.99 to 10.00, SG'FORE's value will be implementation dependent (and must be computed at run-time). The fact that 'FORE may return implementation dependent values should be taken into consideration by programmers.

Ada Commentary ai-00180-bi.wj downloaded on Tue May 10 13:50:02 EDT 1988

!standard 03.09 (05)
!standard 13.09 (03)
!class binding interpretation 84-01-25
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-06-28
!status work-item 84-06-12
!status received 84-01-25
!references 83-00266, 83-00339
!topic Elaboration checks for INTERFACE subprograms

!summary 84-09-07

If a subprogram is named in an INTERFACE pragma, no check need be made that the subprogram body has been elaborated before it is called.

!question 84-06-12

Is the body of a subprogram named in an INTERFACE pragma elaborated at the site of the pragma, or is such a check simply to be omitted?

!recommendation 84-07-17

If a subprogram has been named in an INTERFACE pragma, no check need be made that its body has been elaborated before the subprogram is called.

!discussion 84-09-07

3.9(5) says, "For a subprogram call, a check is made that the body of the subprogram is already elaborated." The use of pragma INTERFACE, however, specifies that no Ada body will be provided and so none can be elaborated. 3.9(5) therefore seems to imply that an attempt to call any subprogram named in an INTERFACE pragma will raise PROGRAM_ERROR. Raising PROGRAM_ERROR would clearly violate the intended purpose of the INTERFACE pragma by making it useless. So it is reasonable (and sufficient) to conclude that no elaboration check need be performed for a subprogram named in an INTERFACE pragma. (An implementation is allowed to perform such a check if it wishes. If such checks are performed, the implementation should specify (in Appendix F) the point at which the body of the subprogram is considered to be elaborated.)

Ada Commentary ai-00181-bi.wj downloaded on Tue May 10 13:50:04 EDT 1988

!standard 04.10 (05) 86-07-23 AI-00181/04
!class binding interpretation 84-01-25
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-06-28
!status work-item 84-06-12
!status received 84-01-25
!references 83-00267
!topic NUMERIC_ERROR for nonstatic universal operands

!summary 84-12-10

When evaluating a nonstatic universal expression, NUMERIC_ERROR can be raised if any operand or the result is a real value that lies outside the range of safe numbers of the most accurate predefined floating point type (excluding universal_real) or an integer value that lies outside the range SYSTEM.MIN_INT .. SYSTEM.MAX_INT.

!question 84-09-10

4.10(5) allows NUMERIC_ERROR to be raised for nonstatic universal expressions only if the RESULTS for the expressions exceed certain values -- NUMERIC_ERROR cannot be raised if the OPERANDS exceed these values. Now consider an expression such as:

```
10E100 mod INTEGER'POS(X)
```

The result of this expression will always be less than INTEGER'LAST, so NUMERIC_ERROR cannot be raised, but run-time evaluation appears to be required if X is nonstatic. Is it acceptable for NUMERIC_ERROR to be raised in such cases?

!recommendation 84-12-10

For the evaluation of an operation of a nonstatic universal expression, an implementation is allowed to raise the exception NUMERIC_ERROR if any operand or the result is a real value whose absolute value exceeds the largest safe number of the most accurate predefined floating point type (excluding universal_real), or an integer value greater than SYSTEM.MAX_INT or less than SYSTEM.MIN_INT.

!discussion 84-09-10

The problem is not limited to integer types. It is also possible to construct nonstatic universal real expressions whose result lies within the range of safe numbers of the largest non-universal real type but with an operand whose value lies outside this range. For example,

```
LARGE/(2.0**INTEGER'POS(X))
```

is one such expression, where LARGE can be a named number whose value lies outside the range of safe numbers and a nonstatic X is chosen so the result lies within that range.

In general, under the current wording, it will sometimes be necessary to evaluate (at run-time) expressions involving values that cannot necessarily be represented exactly using the predefined numeric types. The Standard does not allow NUMERIC_ERROR to be raised in such cases, although it was not the intent to require the run-time representation of values that exceed the ranges of the largest real and integer types.

In order to allow efficient and reasonable run-time implementations for evaluating nonstatic universal expressions, it was the intent to allow NUMERIC_ERROR to be raised when values involved in nonstatic universal expressions exceeded the range of values for the largest supported integer or real type. Therefore, the wording in 4.10(5) should be understood to mean that NUMERIC_ERROR can be raised when a value of a result OR OPERAND is outside the range of such types.

Ada Commentary ai-00186-bi.wj downloaded on Tue May 10 14:45:02 EDT 1988

!standard 13.01 (06) 86-12-04 AI-00186/08
!standard 02.08 (09)
!class binding interpretation 84-03-13
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-09-11 (reviewed)
!status committee-approved (9-0-1) 85-09-05 (pending editorial review)
!status work-item 85-08-01
!status received 84-03-13
!references AI-00039, AI-00321, AI-00322, 83-00297, 83-00496, 83-00532
!topic Pragmas recognized by an impl do not force default representation

!summary 86-12-04

The intent of 2.8(9) is that an invalid pragma have the same effect as if it were absent. To ensure that this intent is realized, a pragma defined by the Standard or by an implementation is not allowed to contain an occurrence of a name or expression that forces the determination of the default representation of a type, since such occurrences would make later representation clauses for the type illegal. Consequently, a representation clause for a type can be accepted even if the clause is given after a pragma that contains an expression that normally would force the default representation of the type to be determined (since such a pragma will be considered invalid, and ignored). (See AI-00322 for a similar rule for pragmas whose identifiers are not defined either by the Standard or by the implementation.)

!question 85-08-01

13.1(6) says with respect to forcing occurrences:

> In any case, an occurrence within an expression is always forcing.

13.1(6) is unequivocal. It makes no exception for expressions that occur in pragmas or for expressions that are not evaluated (e.g., expressions defining default initial values for record components). On the other hand, 2.8(9) says:

> a pragma (whether language-defined or implementation-defined) has no effect if its placement or its arguments do not correspond to what is allowed for the pragma.

These requirements appear to conflict. For example, consider:

```
type YES is new INTEGER range 1..10;
pragma PRIORITY (YES(4));          -- (1)
pragma OPTIMIZE (YES(4)+5);        -- (2)
pragma DEBUG(YES(1));              -- (3)
for YES'SIZE use 5;                -- legal?
```

If the occurrence of YES in (1) is considered a forcing occurrence (as

13.1(6) seems to say), then the representation clause is illegal (13.1[7)]. But the argument to PRIORITY must have type INTEGER, and YES(4) does not have this type. Since the argument is therefore not "what is allowed" for the pragma, the pragma has no effect [2.8(9)]. Certainly the pragma has an effect if the later occurrence of a representation clause is considered illegal, so it would appear that 2.8(9) requires that the pragma, and in particular, the occurrence of YES in the expression be ignored. In short, is the occurrence at (1) forcing or not?

In (2), the type name YES again appears in what appears to be an expression, but OPTIMIZE does not even allow an expression as an argument. Is this occurrence of YES to be considered a forcing occurrence? Is the occurrence of the expression even to be considered legal?

Finally, in (3), let's assume DEBUG is an implementation-defined pragma that expects an argument having some integer type. Let's assume that even a non-static expression is allowed, so the argument YES(1) is an acceptable argument. In this case, is the occurrence of YES a forcing occurrence that makes the representation clause illegal?

!recommendation 86-08-08

If a type's representation has not yet been fully determined (either explicitly or by default), a pragma whose identifier is recognized by an implementation is not allowed to contain an occurrence of a name or expression that forces determination of the default representation of the type. (Since a pragma violating this rule has no effect, no occurrence of a name or expression in a pragma causes the default representation of a type to be determined.)

!discussion 86-09-12

The intent of 2.8(9) is that an invalid pragma have the same effect as if it were not present. In particular, the occurrence of a name or expression in an ignored pragma should not be considered to force determination of a type's default representation, and therefore, a later representation clause for the type can be obeyed. If this were not the case, the intent of 2.8(9) would not be satisfied.

A representation clause for a type is not allowed after the type's default representation has been determined. Certain occurrences of names and expressions cause the default representation of a type to be determined:

- the occurrence of a type name that satisfies the definition of "forcing occurrence" (13.1(6)); the denoted type's representation is determined.

- the operand of a relational operator, type conversion, or membership test (AI-00039); the representation of the operand's type is determined.

- the name of an array type or type having a subcomponent of an

array type; the representations of the index subtypes are determined (AI-00321).

Note that although many occurrences of a type name satisfy the definition of "forcing occurrence," only the first such occurrence actually causes a default representation to be determined.

Of the particular examples given in the question, example (2) is the easiest to resolve, since the pragma OPTIMIZE requires a name for its argument, not an expression. Since the expression does not conform to what is allowed for the pragma, the pragma must have no effect; the occurrence of YES within the expression must not be considered a forcing occurrence. (Moreover, the occurrence of an expression instead of a name is not grounds for saying that the program containing the pragma is illegal, since considering the whole program illegal would also be "an effect" associated with a pragma that is to have no effect.) In short, when a pragma requires a name instead of an expression as its argument, the pragma is ignored if an expression is given. Consequently, occurrence of the expression does not force the default representation of any type to be determined.

So now we only need to consider pragmas that allow expressions as arguments. The only language defined pragmas that allow expressions as arguments are MEMORY_SIZE, PRIORITY, and STORAGE_UNIT. In the case of MEMORY_SIZE and STORAGE_UNIT, the expression must be a literal. Occurrence of any other form of expression or occurrence of any name means the pragma must have no effect. In the case of PRIORITY, the only requirement is that its argument be a static expression whose value belongs to subtype SYSTEM.PRIORITY. A static expression can contain a type name or a relational operator, but it cannot contain any of the other forms that force selection of a default representation. For example, YES'POS(1) or BOOLEAN'POS(PROC."="(1.0, 1.0)) are acceptable arguments for PRIORITY, assuming the value 1 lies in the range of subtype SYSTEM.PRIORITY. If the value 1 is not acceptable, the pragma is ignored, and although the occurrence of YES, for example, satisfies the definition of a forcing occurrence, it would not be consistent with 2.8(9) to consider that the default representation of YES is determined. What if the value 1 is acceptable? Is the representation of any type determined? What if the acceptability of the argument cannot be determined at compile-time? (For example, SYSTEM.PRIORITY need not have a static subtype (AI-00045), so in principle, it is not always possible to decide at compile-time whether a particular priority value is acceptable or not. The same problem arises for any implementation defined pragma that allows non-static expressions as arguments. If the acceptability of the pragma's argument cannot be determined at compile-time, how can one determine if the pragma is to be ignored and hence, if a later representation clause is allowed?)

Several alternatives were considered in deciding how to resolve these issues:

1) occurrences in pragma expressions force a representation to be determined if the pragma is accepted by an implementation.

2) if a pragma's identifier is recognized but is not accepted for some reason, occurrences in the pragma's expression may, but need not

force a representation to be determined (i.e., the effect is implementation defined).

3) occurrences in pragma expressions never force a representation to be determined.

4) do not allow a pragma expression that would cause the default representation of a type to be determined, e.g., the use of a type name in a pragma expression would not be allowed if the type's representation was not yet determined. Any such use would mean the pragma is to be ignored.

5) occurrences in pragma expressions always force a representation to be determined, even if the pragma is otherwise ignored.

Since there seems to be no real need to allow the use of type names or relational operators in pragma expressions, the cleanest solution seems to be solution 4, i.e., a pragma must be ignored if an expression in the pragma would otherwise cause the default representation of a type to be determined. This rule can be supported uniformly by every implementation without imposing a burden on programmers. It is consistent with the intent of ensuring that ignored pragmas have the same effect as if they were absent. It is superior to solution 3 which would simply contradict 13.1(6). It is superior to solutions 1 and 2, which are implementation dependent in their effects. It is superior to solution 5, which contradicts 2.8(9).

(Note that the recommendation allows:

```
pragma PRIORITY(PRIORITY'FIRST);
```

since this occurrence of the name, PRIORITY, is not a forcing occurrence; the representation of PRIORITY's base type has already been determined in STANDARD. Similarly, the use of any type name is allowed in a pragma expression after the representation of the type has been determined.)

Ada Commentary ai-00187-bi.wj downloaded on Tue May 10 14:45:03 EDT 1988

!standard 04.01.03 (15) 87-09-12 AI-00187/06
!class binding interpretation 84-05-14
!status approved by WG9/AJPO 87-07-30 (corrected in accordance with AI-00468)
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-06-28
!status work-item 84-05-14
!status received 84-03-13
!references AI-00468, 83-00301, 83-00343, 83-00802
!topic Using a name decl by a renaming decl as a selector in an expanded name

!summary 84-05-14

A name declared by a renaming declaration can be used as a selector in an expanded name.

!question 84-09-10

Can a name declared by a renaming declaration be used as the selector in an expanded name? The current wording seems to preclude such use since it says the selector of an expanded name must be the name of an entity, and a renaming declaration does not declare an entity.

!recommendation 84-05-14

For a prefix denoting a package, the selector in an expanded name must be a simple name, character literal, or operator symbol declared in the visible part of the package.

For a prefix denoting a program unit, a block statement, a loop statement, or an accept statement, the selector in an expanded name must be a simple name, character literal, or operator symbol declared immediately within the construct denoted by the prefix.

!discussion 87-09-12

A renaming declaration does not declare an entity; it declares "another name for an entity [8.5(1)] (see also [3.1(1)]). The rules in 4.1.3 for expanded names require that the selector of an expanded name be a:

> "simple name, character literal, or operator symbol OF [AN] ENTITY ... declared in the visible part of [the] package [denoted by the prefix]" [4.1.3(15)]
>
> "simple name, character literal, or operator symbol OF AN ENTITY whose declaration occurs immediately within the construct [denoted by the prefix]" [4.1.3(17)]

Since a renaming declaration does not declare an entity, the implication appears to be that the name declared by a renaming declaration can never be used as a selector in an expanded name:

```
package P is
     A : INTEGER;                    -- declares an entity.
end P;

with P;
package Q is
     B : INTEGER renames P.A;        -- B is not an entity.
     -- ok; A is the name of an entity declared in P
end Q;

with Q;
package R is
     C : INTEGER renames Q.B;
     -- illegal? the B in Q is not the name of an entity declared in Q.
end R;
```

This implication does not conform to the intent of the language design. The rules in 4.1.3(15) and 4.1.3(17) should be understood to allow a name declared by a renaming declaration to be used as a selector in an expanded name.

When such names are allowed, an expanded name can contain an indefinite number of simple names:

```
package FRED is
     A : INTEGER;
     package JIM renames FRED;
end FRED;
```

We can then refer to FRED.A as FRED.JIM.A or FRED.JIM.JIM.A, or in general, use as many .JIM's as we like (since both A and JIM are declared in the visible part of a package and therefore can be denoted by an expanded name, even when the prefix is declared by a renaming declaration; see AI-00016. FRED.JIM.A would be illegal if A were not declared in FRED's visible part; similarly FRED.JIM.JIM would be illegal if JIM were not declared in FRED's visible part). Writing USE FRED.JIM has the same effect as writing USE FRED.

Although these forms of expanded name are unusual, they are a natural consequence of allowing a name declared by a renaming declaration as a selector in an expanded name and pose no difficulty for a careful implementer.

Ada Commentary ai-00190-bi.wj downloaded on Tue May 10 14:50:01 EDT 1988

| !standard 04.09 (02) 86-07-23 AI-00190/05
| !standard 04.03.02 (03)
| !class binding interpretation 84-03-13
| !status approved by WG9/AJPO 86-07-22
| !status approved by Director, AJPO 86-07-22
| !status approved by WG9/Ada Board 86-07-22
| !status approved by Ada Board 86-07-22
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-06-28
!status work-item 84-05-24
!status received 84-03-13
!references 83-00286, 83-00376, 83-00378, 83-00379, 83-00454
| !topic A static expression cannot have a generic formal type

!summary 84-09-03

A static expression is not allowed to have a generic formal type (including a type derived from a generic formal type, directly or indirectly). (Consequently, if an array's index subtype is a generic formal type, aggregates for that dimension of the array can have only a single component association and this component association must have a single choice.)

!question 84-09-03

Consider this example:

```
generic
     type T is range <>;
package PACK is
     type ARR is array (T) of BOOLEAN;
     X : ARR := (1_000_000 => FALSE,
                 1_000_001 => TRUE);      -- legal?
end PACK;
```

4.3.2(3) requires that when more than one component association is present in an array aggregate, all the choices must be static. Consider the aggregate (1_000_000 => FALSE, 1_000_001 => TRUE). This aggregate is illegal if 1_000_001 does not belong to the index base type because evaluation of the literal (i.e., its implicit conversion to the index base type) will raise an exception [4.9(2)]. If the index base type is a generic formal type, it is not known at compile time whether 1_000_001 belongs to the base type or not (as long as 1_000_001 does not exceed SYSTEM.MAX_INT). If 1_000_001 belongs to the base type for an instantiation, no exception will be raised. Does the use of such an aggregate in a generic unit mean the legality of the unit's instantiation depends on the range of the base type used in the instantiation?

!recommendation 84-09-03

A static expression is not allowed to have a generic formal type (including a type derived from a generic formal type, directly or indirectly).

!discussion 84-09-03

4.9 specifies the conditions under which an expression can be considered to be static. If an expression has a formal generic type, it can only be static if the formal type is an integer or real type and the primaries in the expression are either numeric literals, named numbers, function calls as specified in 4.9(7), or static expressions enclosed in parentheses. There are no other possibilities. In particular, there are no enumeration literals for a formal discrete type, and since 4.9(11) says a generic formal type is not static, no constants satisfying 4.9(6) can be declared; moreover, there are no static attributes for formal generic types, and a formal generic type cannot be used in a static qualified expression.

In short, an expression such as 1 + 2 or 1.0 + 2.0 can apparently have a generic formal type and still be considered static according to 4.9. Can such expressions be used anywhere static expressions are required? An examination of the Standard shows that rules requiring static expressions are given in the following sections: 3.2.2(1), 3.5.4(3), 3.5.7(3), 3.5.9(3), 3.6.2(2), 3.7.3(3), 4.3.1(2), 4.3.2(3), 4.9(6, 11), 5.4(4), 9.8(1), 13.2(5, 12), and 13.3(4). These rules generally exclude static expressions that have a generic formal type because:

1. A static expression is explicitly required to have a universal type: 3.2.2(1), 3.6.2(2), 13.3(3,4);

2. The type of a static expression is explicitly required to be INTEGER: 9.8(1);

3. An expression that must be static must have some integer or real type that is determined independent of the context of the expression. Since universal integer or universal real is always acceptable in such contexts, expressions having the form 1 + 2 or 1.0 + 2.0 can never be considered to have a non-universal type [4.6(15], and so cannot have a generic formal type: 3.5.4(3), 3.5.7(3), 3.5.9(3), 13.2(5, 12);

4. Expressions having a generic formal type are excluded because a generic formal type is not static: 4.9(6, 11);

5. Generic formal types are explicitly excluded as the type of a discriminant or as the expression in a case statement, and so expressions having such types are not allowed as choices in variant records, as discriminants of record aggregates, or as choices in case statements: 3.7.3(3), 4.3.1(2), 5.4(4).

The only case remaining, choices in array aggregates [4.3.2(3)], has no explicit or implicit exclusion regarding generic formal types. However, since generic formal types are explicitly excluded by 3.7.3(3) and 5.4(4), and since these are cases where allowing a generic formal type would mean the legality of a generic instantiation would depend on the subtype used in its instantiation, it is reasonable to conclude that the intent was to ensure the legality of array aggregates would also be determined prior to instantiating a generic unit.

The required legality check can be ensured in two ways: 1) by not allowing a choice expression in an array aggregate to have a formal generic type; or 2) by saying that any expression having a generic formal type is not static. The latter rule would allow aggregates such as (1_000_000..1_000_001 => TRUE), since when there is a single choice, it is not required to be static [4.3.2(3)]. If an array had an index type whose base type was a generic formal type, the first alternative would forbid any choices for that dimension except OTHERS, since OTHERS is not an expression. It would even forbid T'FIRST..T'LAST, where T is the formal generic type.

Since the intent is to ensure instantiations containing aggregates are legal if the generic template is legal, and since the second alternative is less restrictive than the first and accomplishes the intent, the second alternative is chosen as the recommended interpretation.

Ada Commentary ai-00193-ra.wj downloaded on Tue May 10 14:50:03 EDT 1988

!standard 08.07 (08) 86-07-23 AI-00193/05
!class ramification 85-02-04
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-05-13
!status committee-approved (7-0-2) 85-02-27
!status work-item 85-02-04
!status received 84-03-13
!references 83-00288
!topic The value of 'FIRST's argument in overloading resolution

!summary 85-02-04

Any overloaded identifiers occurring in the argument for 'FIRST(N), 'LAST(N), and 'RANGE(N) must be resolved independently of the context in which these attributes are used.

!question 85-04-01

Consider the following example:

```
procedure P is
     type E1 is (A, B);  -- A < B is true
     type E2 is (B, A);  -- A < B is false
     -- BOOLEAN'POS(A < B) is static and has type universal integer
     -- but it has two interpretations, depending on the
     -- resolution of A and B. This means the attribute potentially
     -- has the value 0 or 1.

     IV  : INTEGER;
     ARR : array (INTEGER range 0..1, BOOLEAN) of FLOAT;

     procedure PUT (X : BOOLEAN) is begin ... end;
     procedure PUT (X : INTEGER) is begin ... end;
begin
     IV := ARR'FIRST(BOOLEAN'POS(A < B));      -- (1) illegal?
     IV := ARR'FIRST(1 + BOOLEAN'POS(A < B)); -- (2) illegal?
     PUT (ARR'FIRST(1));                       -- (3) legal?
end;
```

The name ARR'FIRST(...) must have the type INTEGER for the assignment statements to be legal. Does this mean the types of A and B must be resolved so the argument of FIRST has the value 1? Is the argument of PUT ambiguous or is its type definitely INTEGER?

!response 85-05-27

8.7(8) says "any rule that requires a name or expression to have a certain type, or to have the same type as another name or expression" is used to resolve an overloaded identifier. Since the attribute ARR'FIRST(...) is a

name, this means that the type of IV can be used to help resolve the type of ARR'FIRST, if ARR'FIRST is considered to be an overloaded entity. However, an attribute is not one of the operations for which overloading is defined (see 8.7(1)). This means, in essence, that the type of ARR'FIRST(...) must be determined independent of its context, and in particular, the type required by the context cannot be used to resolve any overloadings contained in its argument.

Hence, with respect to the question, statements (1) and (2) are illegal because the type of the operands in the expression A < B cannot be determined uniquely. Statement (3) is legal because the argument of ARR'FIRST is not overloaded; ARR'FIRST(1) has the type INTEGER.

Ada Commentary ai-00196-ra.wj downloaded on Tue May 10 15:45:01 EDT 1988

!standard 09.06 (06) 86-07-23 AI-00196/05
!class ramification 84-11-28
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-26
!status work-item 84-11-01
!status received 84-03-13
!references 83-00295, 83-00428, 83-00429
!topic Use of 86_400.0 in TIME_OF

!summary 85-01-06

If TIME_OF is called with a seconds value of 86_400.0, the value returned is equal to TIME_OF for the next day with a seconds value of 0.0. In addition, the SECONDS function always returns a value less than 86_400.0, even if the SECONDS argument of TIME_OF was 86_400.0.

!question 85-01-04

Consider:

```
NOW1 := TIME_OF (1984, 12, 9, 86_400.0);     -- call 1
NOW2 := TIME_OF (1984, 12, 10, 0.0);         -- call 2
```

(1) Is an implementation allowed to raise TIME_ERROR for call 1 if the SECONDS parameter is not strictly less than 86,400? If not,

(a) Is it required that NOW1 = NOW2 (since both represent the same instants of time)?

(b) Does the value of DAY (NOW1) = 9 or does DAY (NOW1) = 10? Is the result for DAY implementation dependent? Similarly, does SECONDS (NOW1) = 0.0 or 86,400.0?

!recommendation 85-01-04

No exception is raised. NOW1 = NOW2, DAY (NOW1) = 10, and SECONDS (NOW1) = 0.0.

!discussion 85-01-04

In some military usage, both 2400 and 0000 are considered valid notations for the same instant of time. Therefore, TIME_ERROR should not be raised if 86_400.0 is used as the value for SECONDS in a call to TIME_OF, and similarly, 0.0 should be allowed. In the example, the TIME_OF arguments in the two calls give two notations for the same instant of time, which is canonically represented by the arguments in the second call. Therefore, NOW1 = NOW2 yields TRUE. It was also the intent that DAY and SECONDS return canonical values. Therefore, DAY (NOW1) must return 10, and SECONDS (NOW1) must return 0.0.

!standard 09.08 (00) 86-12-01 AI-00197/07
!class ramification 86-01-21
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-09-11 (reviewed)
!status committee-approved 86-05 (pending editorial review)
!status committee-approved (7+1-0-3) 86-05 (by ballot)
!status committee-approved (6-2-0) 86-02-20 (pending letter ballot)
!status work-item 86-01-21
!status received 84-03-13
!references 83-00299
!topic With SYSTEM clause not needed for pragma PRIORITY

!summary 86-01-21

Use of the pragma PRIORITY does not require the package SYSTEM to be named in a with clause for the enclosing compilation unit.

!question 86-01-21

Does the use of the pragma PRIORITY require the package SYSTEM to be named in a with clause for the enclosing compilation unit?

Such a requirement is explicitly mentioned in 13.5(3) for the use of an address clause, but not for the pragma PRIORITY. Nevertheless, both the type ADDRESS and the subtype PRIORITY are declared in SYSTEM.

!response 86-08-08

PRIORITY is declared in package SYSTEM as a subtype of STANDARD.INTEGER. Since the scope of STANDARD.INTEGER includes any unit being compiled (8.6(2)) and since the pragma PRIORITY requires an argument of type INTEGER (9.8(1)), a value or expression having type INTEGE can be used in the pragma whether or not the subtype PRIORITY is visible. Hence, a with clause for SYSTEM is not needed for a PRIORITY pragma such as:

```
pragma PRIORITY (3);
```

The with clause is not needed even though the pragma has no effect if 3 does not belong to the subtype SYSTEM.PRIORITY. However, a with clause for SYSTEM would be needed for:

```
pragma PRIORITY (SYSTEM.PRIORITY'LAST);
```

since otherwise the names SYSTEM and PRIORITY would not be visible.

The situation is different for ADDRESS. ADDRESS is declared as a type, not a subtype, and the type ADDRESS is declared in SYSTEM, not STANDARD, so the scope of the ADDRESS type does not normally include a unit being compiled.

For the call, TIME_OF (2099, 12, 31, 86_400.0), TIME_ERROR will be raised, since the parameter values specify a year value that is not in the range of YEAR_NUMBER.

Ada Commentary ai-00197-ra.wj downloaded on Tue May 10 15:45:03 EDT 1988

Ada Commentary ai-00199-bi.wj downloaded on Tue May 10 15:50:01 EDT 1988

!standard 10.01 (06) 86-07-23 AI-00199/08
!class binding interpretation 84-12-26
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (8-0-0) 85-05-18
!status work-item 85-02-04
!status received 84-03-13
!references AI-00266, AI-00225, 83-00304, 83-00480, 83-00481, 83-00482, 83-00578
!topic Implicit declaration of library subprograms

!summary 85-09-19

A subprogram body given in a compilation unit (following the context clause) is interpreted as a secondary unit if the program library already contains a subprogram declaration or generic subprogram declaration having the same identifier as the body. Otherwise, the subprogram body is interpreted as a library unit.

Successfully compiling a subprogram body that is a library unit means the unit is added to the library, replacing any previously existing library unit having the same identifier. (The previously existing library unit can only be a package declaration, a generic package declaration, a generic instantiation, or a previously compiled subprogram body that is a library unit.)

!question 85-09-19

Suppose the following library unit is compiled into an empty library:

```
with PACK; use PACK;
procedure P (X : INTEGER) is
begin ... end P;
pragma INLINE(P);
```

(1) Can we now compile a subprogram body P with a different parameter and result type profile, e.g.,

```
procedure P (X : BOOLEAN) is  -- illegal?
```

(2) Is the INLINE pragma ignored since it follows a subprogram body (a secondary unit) instead of a subprogram declaration (a library unit) (see 6.3.2(2))?

(3) If the subprogram body is later recompiled with a new context clause, does the old context clause still apply to the new body?

```
with NEW_PACK; use NEW_PACK;
procedure P (X : INTEGER) is begin ... end P;
```

Now suppose that a generic subprogram or package instantiation is compiled. What is the effect of compiling a new body with the name and profile of the instantiation? In particular, consider this sequence of compilation units:

```
-- Unit 1: A library unit that is a generic subprogram
     generic
     procedure GPROC;

-- Unit 2: Its body
     procedure GPROC is
     begin ... end;

-- Unit 3: An instantiation as a library unit
     with GPROC;
     procedure NEW_PROC is new GPROC;

-- Unit 4: Attempt to recompile a body for NEW_PROC
     procedure NEW_PROC is
     begin ... end;
```

(4) Is Unit 4 legal?

(5) If so, what is the effect if GPROC's body is later compiled?

(6) Must the parameter and result type profile for Unit 4 be the same as the profile created by Unit 3?

(7) Suppose GPROC were a generic package and Unit 4 was an attempt to compile a new body for the instantiated package. Would Unit 4 be legal?

!recommendation 85-09-19

A subprogram body given in a compilation unit (following the context clause) is interpreted as a secondary unit if the program library already contains a subprogram declaration or generic subprogram declaration with the same identifier; it is otherwise interpreted as a library unit.

!discussion 85-10-02

The syntax in 10.1(2) allows a subprogram body to be used either as a library unit or as a secondary unit. This provides the same facility for library units as exists for the declaration of subprograms: the subprogram specification part of the subprogram body can serve as the declaration, obviating the need for a separate subprogram declaration. (Alternatively, a separate declaration can be retained if desired.)

10.1(6) tells how to distinguish the two cases:

> A subprogram body given in a compilation unit is interpreted as a secondary unit if the program library already contains a library unit that is a subprogram with the same name; it is otherwise interpreted both as a library unit and as the corresponding library unit body

(that is, as a secondary unit).

In the first case, the existing library unit must be a subprogram declaration or a generic subprogram declaration. The result of compilation is then to add to the program library the missing body (or to replace an existing body), check conformance, and note any additional with clauses in the new context clause (these as well as the with clauses given for the declaration will apply to the new body). The subprogram (or generic subprogram) is now represented by two units in the program library: a library unit that is the declaration, and a separate secondary unit with the same identifier that is the corresponding body.

In the second case we are compiling a new subprogram, in the form of a library unit which will serve as both declaration and body. Because library units must have different identifiers, this library unit will replace any existing library unit with the same name -- see 10.1(3, 4). In particular, it will replace a library unit that is a subprogram, and its context clause will supersede the old one.

It should be noted that a generic instantiation is a distinct form of library unit. (The identifier of this library unit is that of the declared instance.) Although a generic subprogram instantiation declares a subprogram -- the generated instance -- it is not itself a subprogram. Likewise, a generic package instantiation is not a package. Although the compiler may at some stage generate the instance, this is distinct from the effects described in 10.1.

The result of an attempt to recompile the body of the (nonexistent) instance follows from the above rule:

For a generic package instantiation, a package body with the same identifier will be rejected; it is a secondary unit, and there is no library unit package declaration with this identifier.

For a generic subprogram instantiation, however, a subprogram body will be treated as a new subprogram library unit and will replace the generic instantiation library unit.

The existing wording of 10.1(6) uses the phrase "library unit that is a subprogram." This is clearly meant to cover the case of an existing declaration, but it is not obvious that it excludes an existing subprogram body serving as both declaration and body. But question (3) shows that if a recompilation of the body with a new context clause is not considered to replace the previously compiled unit (and its context clause), then unintended naming conflicts can arise. In particular, there would be no way to get rid of the potentially obsolete context clause except by explicitly compiling a subprogram declaration for P that contains a different context clause (or none at all). This is the reasoning behind the above interpretation.

With the recommending interpretation, the answers to the questions are:

(1) The new body replaces the old subprogram as a whole. The new subprogram body P is a new library unit, and therefore need not conform to the previously compiled unit's specification.

(2) The INLINE pragma is correctly given after a library unit, and naming that unit. The procedure P is interpreted as a library unit because it is compiled into an empty library, and so the pragma can be obeyed. (The pragma is part of the compilation, but not part of the program library; it can be held in the library file -- see 10.4. A subsequent recompilation of the body will replace the library unit in the program library, but not the pragma, which can still be obeyed.)

(3) Only the new context clause applies to the newly compiled library unit.

(4, 5, 6) Unit 4 is accepted, but does not affect the body of the procedure NEW_PROC because the generic instantiation is a library unit and is not a subprogram; instead, the instantiation is replaced by the new procedure as a library unit.

(7) If GPROC were a generic package and Unit 4 were an attempt to compile a new body for the instantiated package, Unit 4 would be rejected. There would be no package declaration as a library unit for which this unit could serve as body, since the generic instantiation is not itself a package.

Ada Commentary ai-00200-bi.wj downloaded on Tue May 10 15:50:03 EDT 1988

!standard 10.03 (07) 86-12-01 AI-00200/08
!standard 06.03.02 (03)
!class binding interpretation 86-02-20
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved (10-1-2) 86-09 (by ballot)
!status committee-approved (7-1-0) 86-02-20 (pending letter ballot)
!status work-item 85-10-29
!status received 84-03-13
!references 83-00306, 83-00692
!topic Dependences created by inline of generic instantiations

!summary 86-04-02

If inline inclusion of a subprogram call is achieved due to pragma INLINE, an implementation is allowed to create a dependence of the calling unit on the subprogram body; when such a dependence exists, the unit containing the call is obsolete if the subprogram body is obsolete. Such dependences can be created even when the subprogram is created as a result of a generic instantiation.

!question 86-03-11

Can the INLINE pragma be obeyed in the following cases?

```
package P is
    generic function GF1 return INTEGER;
    generic function GF2 return INTEGER;
    pragma INLINE (GF1);
end P;

package body P is
    function GF1 return INTEGER is ... end GF1;
    function GF2 return INTEGER is ... end GF2;
end P;

with P;
package R is
    function F1 is new P.GF1;
    function F2 is new P.GF2;
    pragma INLINE (F2);
    X : INTEGER := F1;          -- is inline inclusion ok? (yes)
    Y : INTEGER := F2;          -- is inline inclusion ok? (yes)
end R;
```

A superficial analysis based on 6.3.2 might suggest that the INLINE pragma can be obeyed for both calls, but what if the body of P is recompiled? 10.3(7) does not allow the recompilation of P's body to make R obsolete. 10.3(7) says:

> If a pragma INLINE is applied to a subprogram declaration given

> in a package specification, ... inline inclusion creates a dependence of the calling unit on the package body, and the compiler must recognize this dependence when deciding on the need for recompilation.

Since a generic instantiation is not (syntactically) a subprogram_ declaration, no dependence on P's body can be created, so it seems an implementation could only generate inline code for calls of F1 and F2 if recompilation of P's body did not make R obsolete (e.g., if the inline code in R were replaced "behind the scenes" by the implementation).

Since implementations are unlikely (and perhaps are not allowed) to implicitly recompile R, the net effect of the rule in 10.3(7) seems to be that inline inclusion cannot be safely achieved for instantiations of a generic subprogram declared in the visible part of a package. Was this the intent?

!recommendation 86-06-17

If pragma INLINE is applied to a generic subprogram declaration or to a subprogram declared by a generic instantiation and inline inclusion of a call of an instance is achieved, the inline inclusion creates a dependence of the calling unit on the generic unit's body; if the generic body becomes obsolete, the units containing such calls are obsolete (and must be recompiled, unless no longer needed). Such dependences on generic unit bodies can be created due to pragma INLINE even when the generic body and the calling unit are in separate compilations and even when the subprogram or generic subprogram to which the pragma applies is declared in a generic package specification.

Similarly, if pragma INLINE is applied to a (non-generic) subprogram declaration given in a generic package specification and inline inclusion of a call is achieved for the corresponding subprogram declared in an instance of the package specification, the inline inclusion creates a dependence of the calling unit on the corresponding subprogram body; if the body becomes obsolete (e.g., because the generic unit's package body is recompiled), the units containing such calls are obsolete (and must be recompiled, unless no longer needed). Such dependences on subprogram bodies can be created due to pragma INLINE even when the generic body, subprogram body, and the calling unit are in separate compilations.

!discussion 86-06-17

6.3.2(3) states:

> If the name of a generic subprogram is mentioned in the pragma [INLINE], this indicates that inline expansion is desired for calls of all subprograms obtained by instantiation of the named generic unit.

This clearly indicates an intent that pragma INLINE be useful when applied to generic subprogram declarations or to subprograms declared in generic

packages. It is an oversight that 10.3(7) does not mention generic subprograms and their instantiations, or instances of generic package specifications.

The recommendation treats two cases. First, an inlined call to an instance of a generic unit is allowed to create a dependence on the corresponding generic unit body. Such an inlined call is allowed if the pragma INLINE applies to the instance or to the generic unit whose instantiation creates the instance. Thus inline inclusion of the calls to F1 and F2 (in the question) is allowed, and if achieved, recompilation of package P's body will make package R obsolete.

The wording of the first paragraph of the recommendation also applies if a generic unit or instantiation appears in a generic package specification or if the subprogram body is compiled as a subunit:

```
generic                              -- library unit
package P1 is
     generic
     procedure GP;
     pragma INLINE (GP);
end;

package body P1 is
     procedure GP is separate;
end P1;

separate (P1)                        -- subunit
procedure GP is
begin ... end GP;

with P1;
procedure MAIN is
     package INST is new P1;
     procedure PR is new INST.GP;
begin
     PR;
end MAIN;
```

The effect of the first paragraph of the recommendation is to allow an implementation to create a dependence between MAIN (the calling unit) and subunit GP (the corresponding subprogram body). If subunit GP is recompiled, MAIN is obsolete and must be recompiled. Similarly, MAIN is made obsolete by the recompilation of package body P1, since such a recompilation makes GP's body obsolete.

The second paragraph of the recommendation covers the case where a subprogram is not a generic unit but appears in a generic package specification. If GP is not a generic unit and MAIN contains a call of INST.GP, recompilation of subunit GP or package body P1 would imply procedure MAIN is obsolete and must be recompiled.

Ada Commentary ai-00205-ra.wj downloaded on Tue May 10 16:45:01 EDT 1988

!standard 03.05.07 (06) 87-06-18 AI-00205/06
!class ramification 84-03-13
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
!status work-item 86-07-30
!status received 84-03-13
!references 83-00281, 83-00493
!topic The formula for MANTISSA is correct

!summary 87-06-04

The number of mantissa bits for D decimal digits of accuracy is correctly given by the formula in the Standard, namely, the integer next above (D*log(10)/log(2)) + 1.

!question 87-06-04

The Standard says that the number of binary digits in the mantissa of a model floating point number is the integer next above (D*log(10)/log(2)) + 1. This seems to be one digit too many, since to ensure D digits of accuracy, it is sufficient if 2**(-B) <= 10**(-D), i.e.:

```
-B * log(2) <= -D * log(10)
B >= D * log(10)/log(2)
B = ceiling(D * log(10)/log(2))          (since B is an integer)
```

This gives a value one less than the value specified in 3.5.7(6). Is the formula given in the Standard correct?

!response 86-07-30

The formula given in the Standard is correct. The Standard says [3.5.7(6)]:

> The number [of binary mantissa bits] B associated with [the minimal number of decimal digits] D is the smallest value such that the relative precision of the binary form is no less than that specified for the decimal form.

This requirement is not equivalent to saying that 2**(-B) must be less than or equal to 10**(-D). An example may be the best way to see the problem. Consider D=2, so that by the proposed rule, B = 7, whereas by the Standard's rule, B = 8. Now consider numbers in the vicinity of 8, whose representation is 0.80E1 in decimal, and 2#0.1000_000#E4 if carried to 7 bits of precision in binary. Now the difference between 8 and the number next above it in the decimal representation is, of course, 0.1, whereas the difference in the binary representation is 2#0.0000_001#E4, or 0.125 (1/8). The relative error for the decimal representation is 0.1/8.0 = 1/80, while the relative error

for the binary form is 0.125/8.0 = 1/64 > 1/80, i.e., the binary form is less accurate than the decimal form, since the relative error is greater, and this is not allowed by 3.5.7(6).

The Standard requires that the largest relative error for model numbers be less than the smallest relative error for the corresponding decimal numbers. (If so, every D-digit decimal number can be represented uniquely as a model number, and such a model number can, in turn, be uniquely mapped back to the original decimal number; see I. B. Goldberg, "27 bits are not enough for 8-digit accuracy," CACM 10, 2 (Feb. 1967), pp. 105-106.)

The maximum relative error for a D-digit number is one divided by the smallest D-digit number, e.g., for D = 2, the maximum relative error is 1/10 = 10**(-D+1). The minimum relative error for a 2-digit decimal number is 1/99, which is slightly larger than 10**(-D), so a lower bound on the minimum relative error is 10**(-D). Corresponding calculations apply to B-digit binary numbers. In short, in order to ensure that every decimal number has a unique model number representation, it is necessary that:

max binary (model) number error <= min decimal number error, i.e.,
2**(-B+1) <= 10**(-D)

This relation leads directly to the formula given in the Standard.

Ada Commentary ai-00210-co.wj downloaded on Tue May 10 16:45:03 EDT 1988

!standard 05.07 (02) 86-12-01 AI-00210/04
!class confirmation 84-03-13
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (9-0-0) 86-05-12 (pending editorial review)
!status work-item 86-03-24
!status received 84-03-13
!references 83-00278, 83-00278
!topic Loop name in an exit statement as an expanded name

!summary 86-04-16

An expanded name is allowed as the loop name in an exit statement.

!question 86-06-17

The syntax of the exit statement in 5.7 specifies an optional syntactic category loop_name. Should this syntactic category not be loop_simple_name? The syntax for loops in 5.5(2) uses loop_simple_name, and the syntactic category, name, in 4.1 subsumes indexed components, slices, etc.

!response 86-08-05

An expanded name is allowed as the loop name in an exit statement. In 5.7(2), "loop" is italicized in the syntactic category loop_name. The italicized part of a syntactic category has a specific meaning as stated in 1.5(11):

> If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic meaning.

Although it is required that the loop name be a simple name at the beginning and at the end of the loop statement, the name in the exit statement can be an expanded name. The expanded name is sometimes needed to reference hidden loop names. For example:

```
procedure T1 is
begin
A:  loop
        ...
T2:     declare
            A : INTEGER := 0;
        begin
            ...
            exit T1.A when ...;  -- (1)
            ...
        end T2;
    end loop A;
```

```
        end T1;
```

Since the loop name A is hidden by the inner declaration, an expanded name must be used in the exit statement at (1).

Since an expanded name is useful, the syntax does not require a simple name to be used in exit statements. Moreover, there is no confusion with the indexed component or slice forms of names since these forms cannot denote a loop name, and so would be illegal.

Ada Commentary ai-00215-bi.wj downloaded on Tue May 10 16:50:03 EDT 1988

!standard 14.03.08 (20) 86-07-23 AI-00215/05
!class binding interpretation 84-05-03
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-04
!status work-item 84-06-12
!status received 84-03-13
!references 83-00276, 83-00352, 83-00354
!topic Type of EXP should be FIELD

!summary 85-10-02

The type of the EXP parameter for PUT is FIELD, not INTEGER.

!question 85-10-02

Shouldn't the type of the EXP parameter for PUT be FIELD so as to be consistent with the declaration given in the specification of the package TEXT_IO in 14.3.10 and the declaration of DEFAULT_EXP given in 14.3.8(6)?

!recommendation 85-10-02

The type of the EXP parameter for PUT is FIELD, not INTEGER.

!discussion 85-09-16

In order to be consistent with the type usually provided for the EXP parameter (e.g., see 14.3.8(12)) and to be consistent with the declarations given in 14.3.10 and 14.3.8(6), the type of the EXP parameter in 14.3.8(20) should be FIELD, not INTEGER.

This commentary is classified as a binding interpretation since although it corrects an obvious typographical error in the Standard, it nonetheless affects the substance of the Standard by resolving an inconsistency.

Ada Commentary ai-00219-bi.wj downloaded on Tue May 10 16:50:04 EDT 1988

!standard 04.09 (02) 86-07-23 AI-00219/06
!class binding interpretation 84-03-13
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-26
!status work-item 84-06-11
!status received 84-03-13
!references 83-00323
!topic Use of & and 'IMAGE in static expressions

!summary 84-12-10

In a static expression, every factor, term, simple expression, and relation must have a scalar type.

!question 84-12-10

According to 4.9(2), the following expressions are static:

```
'a' & 'b' = 'c' & 'd'
INTEGER'IMAGE(555) = INTEGER'IMAGE(666)
```

In the first expression, each primary is an enumeration literal, the expression delivers a scalar type (BOOLEAN), and "&" and "=" are predefined operators. In the second expression, IMAGE is a language defined attribute of a static subtype (INTEGER) and its actual parameter is a static expression, so both primaries satisfy the requirements of 4.9(f). The fact that 'a' & 'b' and INTEGER'IMAGE(555) are not themselves static expressions (because they are not of a scalar type) doesn't seem to matter. The complete expressions given above satisfy the definition of a static expression, even though this is probably not what was intended. Are the above expressions intended to be considered static expressions?

!recommendation 84-12-10

In a static expression, every factor, term, simple expression, and relation must have a scalar type. (This implies that neither the catenation operator nor the predefined attribute IMAGE can be used in static expressions.)

!discussion 84-12-10

The intent of the RM was to require only the compile time evaluation of scalar expressions. The examples show that this intent is not realized if "&" and 'IMAGE are allowed in such expressions.

Ada Commentary ai-00225-bi.wj downloaded on Tue May 10 17:45:01 EDT 1988

!standard 10.01 (06) 86-07-23 AI-00225/09
!class binding interpretation 84-03-13
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status WG9/ADA Board approved (provisional) 85-05-13
!status committee-approved (10-0-0) 85-02-27
!status work-item 85-02-04
!status received 84-03-13
!references AI-00199, 83-00332, 83-00366, 83-00578
!topic Secondary units for generic subprograms

!summary 85-05-27

If a subprogram body given in a compilation unit has the same identifier as a library unit and the library unit is a generic subprogram declaration, then the subprogram body is interpreted as a secondary unit.

!question 84-04-01

10.1(6) says "A subprogram body given in a compilation unit is interpreted as a secondary unit if the program library already contains a library unit that is a subprogram with the same name." As currently phrased, after compiling a generic subprogram declaration, the library contains a generic unit, not a subprogram. Consequently, when a subprogram body is later compiled, no (non-generic) subprogram will be found in the library, and the subprogram body will not be considered to be the body of the generic unit. Surely this was not the intent.

!recommendation 85-05-27

If a subprogram body given in a compilation unit has the same identifier as a library unit and the library unit is a generic declaration for a generic subprogram, then the subprogram body is interpreted as a secondary unit.

!discussion 85-10-31

See the discussion for AI-00199, which covers this situation as well as related situations.

Ada Commentary ai-00226-bi.wj downloaded on Tue May 10 17:45:02 EDT 1988

!standard 10.01.01 (04) 86-07-23 AI-00226/06
!class binding interpretation 84-03-13
!status approved by WC9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-04
!status work-item 84-06-12
!status received 84-03-13
!references 83-00333
!topic Applicability of context clauses to subunits

!summary 85-09-16

The context clause of a library unit that is a declaration applies not only to the secondary unit that defines the corresponding body, but also to any subunits of the secondary unit.

!question 85-09-16

10.1.1(4) says:

> The with clauses and use clauses of the context clause of a library unit apply to this library unit and also to the secondary unit that defines the corresponding body. ... Similarly, the with clauses and use clauses of the context clause of a compilation unit apply to this unit and also to its subunits, if any.

According to this paragraph, the context clause of a library unit that is a declaration (e.g., a package specification or subprogram declaration) does not apply to the subunits of its body (since the declaration and the body are two different compilation units). This, however, is inconsistent with 10.2(6) which states:

> Visibility within the proper body of a subunit is the visibility that would be obtained at the corresponding body stub (within the parent unit) if the with clauses and use clauses of the subunit were appended to the context clause of the parent unit.

Shouldn't the context clause for a library unit that is a declaration also apply to the subunits of its secondary unit, if any?

!recommendation 85-04-10

The with clauses and use clauses that apply to a secondary unit also apply to its subunits, if any.

!discussion 84-06-12

Consider the following set of compilation units:

Ada Commentary ai-00350-ra.wj downloaded on Thu May 12 08:45:02 EDT 1988

AD A197 564

!standard 06.03.01 (05) 86-12-04 AI-00350/04
!standard 02.10 (05)
!class ramification 86-04-14
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (8-1-0) 86-05-12 (pending editorial review)
!status work-item 86-04-10
!status received 85-06-18
!references 83-00554
!topic Lexical elements not changed by allowable character replacements

!summary 86-04-14

Lexical elements differing only in their use of allowable replacements of characters (as defined in 2.10) are considered as the same. In particular, use of the allowable replacements does not affect the conformance of formal parts, discriminant parts, or actual parameters.

!question 86-06-26

Is a replacement allowed by 2.10 a sufficient change to form a different lexical element? For example:

```
procedure P(S : String := (1 | 2 | 3 => 'x')); -- P1
-- Vertical bars in aggregate

procedure P(S : String := (1 | 2 ! 3 => 'x'))  -- P2
-- Exclamation mark in aggregate
is ... end P;

procedure Q(S : String := "Foo%bar");          -- Q1
-- Double quotes in string

procedure Q(S : String := %Foo%%bar%)          -- Q2
-- Percent signs in string
is ... end Q
```

Although the inconsistent usage of replacement of characters is not good programming practice, do P2 and Q2 conform to P1 and Q1, respectively?

!response 86-12-04

6.3.1(5) states:

> Two subprogram specifications are said to conform if, apart from [certain variations allowed by 6.3.1(2-4)], both specifications are formed by the same sequence of lexical elements, and corresponding lexical elements are given the same meaning by the visibility and overloading rules.

2.10(1-4) defines allowable replacements for the vertical bar, sharp, and quotation characters in programs. 2.10(5) then says:

> These replacements do not cnange the meaning of the program.

In particular, use of the allowed replacements does not change the meaning of a program with respect to the conformance rules, i.e., a legal program cannot become illegal just because a lexical element has been changed by use of an allowed character replacement. Hence, P2 and Q2 conform to P1 and Q1, respectively.

Ada Commentary ai-00354-ra.wj downloaded on Thu May 12 08:45:03 EDT 1988

!standard 10.05 (02) 86-12-01 AI-00354/03
!class ramification 85-06-18
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (8-0-1) 86-05-12 (pending editorial review)
!status work-item 86-04-08
!status received 85-06-18
!references 83-00562, 83-00568
!topic On the elaboration of library units

!summary 86-07-07

There is no requirement that the body of a library unit be elaborated as soon as possible after the library unit is elaborated. In particular, the pragma ELABORATE should be used if it is important that a library package's body be elaborated before another package is elaborated.

!question 86-04-15

10.5(2) says:

> The elaboration of these library units and of the corresponding library unit bodies is performed in an order consistent with the partial ordering defined by the with clauses ...

Does this wording define the order of library unit bodies in the elaboration? Consider the following example:

```
-- unit 1
package P is
    A : INTEGER := 5;
end;

-- unit 2
package body P is
begin
    A := 10;
end;

-- unit 3
with P; use P;
package Q is
    B : INTEGER := A;
end;

with Q; use Q;
procedure R is
begin
    PUT(B);   -- the result of this B
end;
```

Are the following interpretations correct?

a. When the order of elaboration is 1-2-3, the value of the variable B is 10.

b. When the order of elaboration is 1-3-2, the value of the variable B is 5.

!response 86-07-07

There is no requirement that a library unit body be elaborated as soon as possible. In particular P's package body need not be elaborated as soon as possible after P's specification is elaborated, so the orders 1-2-3 and 1-3-2 are both permitted before executing main program R. The pragma ELABORATE should be used if it is important that P's body be elaborated before package Q is elaborated.

Ada Commentary ai-00355-ra.wj downloaded on Thu May 12 08:50:02 EDT 1988

!standard 10.05 (04)
!standard 09.06 (07)
!standard 13.07 (02)
!standard 13.10.01 (01)
!standard 13.10.02 (01)
!standard 14.01 (01)
!standard 14.03 (01)
!standard 14.06 (05)
!standard C (22)
!class ramification 86-02-20
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved 86-02-20 (7-0-1) (pending editorial review)
!status received 85-06-18
!references AI-00325, 83-00553, 83-00586
!topic Pragma ELABORATE for predefined library packages

!summary 86-09-05

An attempt to use an entity declared within a predefined library unit or a unit declared within a predefined library package must raise PROGRAM_ERROR if a required body has not been elaborated (3.9(5-8)). An implementation is not allowed to raise PROGRAM_ERROR for this reason, however, if a pragma ELABORATE has been given for the library unit. (In particular, if the library unit body provided by the implementation depends on other (implementation-defined) library units, the implementation must ensure prior elaboration of the required bodies, e.g., by providing appropriate ELABORATE pragmas.)

!question 86-05-05

Consider the following compilation unit:

```
with TEXT_IO; use TEXT_IO;
pragma ELABORATE (TEXT_IO);
package body IO_UTILITIES is
     package MY_FLOAT is new FLOAT_IO (FLOAT);
     ...
end IO_UTILITIES;
```

When the instantiation of FLOAT_IO is elaborated, the body of FLOAT_IO must also have been elaborated or else PROGRAM_ERROR will be raised (3.9(7-8)). The use of the pragma ELABORATE ensures that the body of TEXT_IO, and hence, the body of FLOAT_IO, will be elaborated before it is accessed. This would seem to prevent PROGRAM_ERROR from being raised, but suppose the body of TEXT_IO has been implemented as follows:

```
with ..., SUPPORT_PACKAGE, ...;
package body TEXT_IO is
     ...
```

```
        package body FLOAT_IO is
             A_CONSTANT : constant ... := SUPPORT_PACKAGE.F(...);
             ...
        end FLOAT_IO;
    end TEXT_IO;
```

Since no pragma ELABORATE is given for SUPPORT_PACKAGE, the body of SUPPORT_PACKAGE need not be elaborated before TEXT_IO's body is elaborated. If SUPPPORT_PACKAGE's body has not been elaborated, the attempted instantiation of FLOAT_IO will raise PROGRAM_ERROR. Since the pragma for TEXT_IO in the compilation of IO_UTILITIES does not ensure the body of SUPPORT_PACKAGE will be elaborated soon enough, is an implementation allowed to raise PROGRAM_ERROR when FLOAT_IO's instantiation is elaborated?

!response 86-09-05

A correct implementation of the Standard must provide all required facilities unless it is impossible or impractical to do so given an implementation's execution environment (AI-00325). In particular, the required predefined library units must be supported. With respect to pragma ELABORATE and the bodies of predefined library units, if a predefined library unit is named in the pragma, a correct implementation must ensure that all needed bodies are elaborated before the named library unit body is elaborated. Otherwise, an attempt to use an entity declared within the library unit (or to use a unit declared within a library package) could raise PROGRAM_ERROR because a needed body has not been elaborated; raising an exception in this case would mean a facility required by the Standard has not been provided.

With respect to the example given in the question, a correct implementation of TEXT_IO's body must include a pragma ELABORATE for SUPPORT_PACKAGE (and any other units whose body should be elaborated before TEXT_IO's body is elaborated).

Ada Commentary ai-00357-bi.wj downloaded on Thu May 12 08:50:03 EDT 1988

!standard 14.02.01 (09) 87-06-18 AI-00357/05
!standard 14.02.01 (15)
!class binding interpretation 85-06-18
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (8-0-1) 86-09-10 (pending editorial review)
!status work-item 86-04-08
!status received 85-06-18
!references 83-00549
!topic CLOSE or RESET of a sequential file from OUT_FILE mode

!summary 86-09-17

If a sequential input-output file having mode OUT_FILE is closed or reset, the most recently written element since the last open or reset is the last element that can be read from the file. If no elements have been written, the closed or reset file is empty. (As a consequence, opening a sequential input-output file with mode OUT_FILE or resetting a sequential input-output file to mode OUT_FILE has the effect of deleting the previous contents of the file.)

!question 86-09-17

Consider the following sequence of actions for a sequential file:

1. Create file for output.
2. Write 10 elements into the file.
3. Reset the file for input.
4. Read in the 10 elements in the file.
5. Reset the file for output.
6. Write 5 elements into the file.
7. Reset the file for input.
8. Read elements until the end of the file.

Step 8 will cause the 5 new elements to be read followed by the last 5 old elements. Shouldn't only the 5 new elements be read?

If we substitute step 5 with:

5a. Close the file.
5b. Open the file for output.

will step 8 cause only the 5 new elements to be read?

!recommendation 86-09-17

CLOSE severs the association between the given file and its associated external file. In addition, for sequential files, if the file being closed

has mode OUT_FILE, the last element written since the most recent open or reset is the last element that can be read from the file. If no elements have been written, the closed file is empty.

RESET repositions the given file so that reading from or writing to the file's elements starts from the beginning of the file. In addition, for sequential files, if the given file has mode OUT_FILE when RESET is called, the last element written since the most recent open or reset is the last element that can be read from the file. If no elements have been written, the reset file is empty.

!discussion 86-09-17

It was the intent that sequential input-output should work for tapes. Sequential input-output is allowed to support files on disk media as long as it models the behavior of tape operations. Consequently, for sequential input-output, writing to the file replaces the previous contents of the file. Therefore, RESET and CLOSE are intended to have the following effect on the contents of the file:

> If the file being reset or closed has mode OUT_FILE, the most recently written element becomes the last element that can be read from the file. If no elements have been written since the file was opened or reset to mode OUT_FILE, then the file is empty after the reset or close.
>
> RESET or CLOSE has no effect on the contents of a file that has mode IN_FILE.

After step 5 in the question, there is no sequence of actions that will allow us to read the 10 elements that were in the file; if no elements are written, the next reset or close will make the file empty.

After step 5a, the file has 10 elements, but step 5b has the effect of deleting these 10 elements, since only newly written elements will be accessible after a subsequent reset or close.

In short, opening a sequential input-output file with mode OUT_FILE or resetting a sequential input-output file to mode OUT_FILE has the effect of deleting the previous contents of the file.

There is no problem with the effect of CLOSE or RESET for direct input-output and text input-output. If the current mode of a direct input-output file is either IN_FILE or OUT_FILE, RESET has the effect of setting the current file index to one so reading from or writing to the file's elements can be restarted from the beginning of the file. The size of the file is not affected. Similarly, CLOSE does not affect the size of a direct file. If the operations in the question were applied to a direct file, step 8 would read 10 elements -- the 5 new elements followed by the last 5 old elements.

For text files, 14.3.1(3) states:

> For the procedure CLOSE: If the file mode has the current mode OUT_FILE, has the effect of calling NEW_PAGE, unless the current page is already terminated; then outputs a file terminator.

14.3.1(4) states:

> For the procedure RESET: If the file has the current mode OUT_FILE, has the effect of calling NEW_PAGE, unless the current page is already terminated; then outputs a file terminator.

For both RESET and CLOSE, a file terminator is output, ensuring that only the elements written since the last open or reset can be read. If the file in the example is a text file, after step 5, the file has 10 elements. After step 7, only the 5 new elements exist in the file so we will read only the 5 new elements at step 8.

Ada Commentary ai-00358-bi.wj downloaded on Thu May 12 09:45:01 EDT 1988

!standard 03.07.02 (05) 86-12-04 AI-00358/10
!standard 03.07 (08)
!class binding interpretation 85-07-07
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (8-0-0) 86-09-10 (pending editorial review)
!status committee-approved (9-0-1) 86-05-12 (pending editorial review)
!status committee-approved 85-09-05 (10-0-0) (pending editorial review)
!status received 85-07-07
!references AI-00007, 83-00574, 83-00576, 83-00585, 83-00731, 83-00734, 83-00786, 83-00798
!topic Discriminant checks for non-existent subcomponents

!summary 86-09-13

When checking the compatibility of a discriminant constraint, 3.7.2(5) requires that a discriminant's value be substituted in component subtype definitions that depend on the discriminant. This substitution is performed only for those subcomponents that exist in the subtype defined by the constraint.

!question 86-07-07

Consider the following declaration of a record type that uses a discriminant's value within one of its variants:

```
type ENUM is (A, B, C);
subtype ENUM_A is ENUM range A..A;
subtype ENUM_AB is ENUM range A..B;

type REC1 (D_A : ENUM_A; POS : POSITIVE) is
    record
        C1 : INTEGER;
    end record;

type REC2 (D_AB : ENUM_AB) is
    record
        case D_AB is
            when A =>
                C2 : REC1 (D_AB, 1);
                C3 : REC1 (D_AB, 0); -- CONSTRAINT_ERROR? (not here)
                C4 : REC1 (A, 0);    -- CONSTRAINT_ERROR? (yes)
            when B =>
                C5 : INTEGER;
        end case;
    end record;

OBJ_B : REC2 (B);                    -- CONSTRAINT_ERROR? (no)
```

Is CONSTRAINT_ERROR raised when the declaration of C3 or C4 is elaborated?

If components C3 and C4 do not exist, should CONSTRAINT_ERROR be raised by the declaration since the discriminant value, B, does not belong to the subtype declared for REC1's discriminant? Or is the substitution required by 3.7.2(5) to be carried out only for those subcomponents that exist in the subtype REC2 (B)?

!recommendation 86-09-13

When checking a discriminant constraint for compatibility (see AI-00007), only components in the subtype defined by the constraint are considered.

!discussion 86-12-04

3.7(8) says:

> For the elaboration of a component subtype definition, if the constraint does not depend on a discriminant (see 3.7.1), then the subtype indication is elaborated. If, on the other hand, the constraint depends on a discriminant, then the elaboration consists of the evaluation of any included expression that is not a discriminant.

When a subtype indication with a constraint is elaborated, the constraint is checked for compatibility (3.3.2(8)). This means that subtype indications are checked if they have a constraint that does not depend on a discriminant. If the constraint does depend on a discriminant, the compatibility check is deferred until a discriminant value is available. With respect to the example given in the question, the elaboration of C4's declaration will raise CONSTRAINT_ERROR because the value 0 does not belong to the subtype for POS and because the constraint does not depend on a discriminant. Elaboration of the subtype definitions for components C2 and C3 consists of evaluating the expressions for the second discriminant; the subtype indication itself is not elaborated, and so no compatibility check is performed.

3.7.2(5) says:

> In addition, for each subcomponent whose component subtype [definition] depends on a discriminant, the discriminant value is substituted for the discriminant in this component subtype [definition] and the compatibility of the resulting subtype indication is checked.

Does checking "each subcomponent" mean checking just those components that exist for a particular subtype? Or does it mean every component declared by a record type definition is to be checked, independent of whether the components can exist when a discriminant has a certain value? The wording is ambiguous, but since no practical value is served by requiring that a check be performed on components that cannot exist for particular discriminant values (indeed, such checks can be counterproductive), it is reasonable to assume that just those subcomponents that exist should be checked. Given this interpretation, an exception should be raised only if a discriminant value is not compatible with its use in declaring a component that exists.

With respect to the example given in the question, if component C4 is deleted from the record type declaration, no exception will be raised when the type declaration is elaborated. When the declaration for OBJ_B is elaborated, no exception will be raised since no compatibility check will be performed for components C2 and C3 (since these subcomponents do not exist in the subtype). Now consider the declaration:

```
OBJ_A : REC2(A);
```

When this declaration is elaborated, CONSTRAINT_ERROR will be raised by the compatibility check for component C3 since the value 0 does not belong to POS's subtype.

Ada Commentary ai-00362-co.wj downloaded on Thu May 12 09:45:03 EDT 1988

!standard 13.07.02 (08) 86-12-01 AI-00362/03
!class confirmation 86-05-13
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (7-0-2) 86-05-13 (pending editorial review)
!status work-item 86-01-27
!status received 85-07-07
!references 83-00569
!topic "component of a record" for representation attributes

!summary 86-07-07

The prefix of 'POSITION, 'FIRST_BIT, or 'LAST_BIT must denote a component of a record object.

!question 86-07-07

Consider the following example:

```
type ARR is array (1 .. 9) of INTEGER;
type REC is record
    A : ARR;
end record;

OBJ : REC;

...
...  OBJ.A(2)'FIRST_BIT ...          -- Legal? (no)
```

Is OBJ.A(2) "a component of a record" for the purposes of 13.7.2(7-10), which says:

> For any component C of a record R:
>
> R.C'POSITION Yields ...
>
> R.C'FIRST_BIT Yields ...
>
> R.C'LAST_BIT Yields ...

That is, is "component" intended to be restrictively interpreted as meaning only immediate components of a record object? (This implies that the prefix of one of the attributes POSITION, FIRST_BIT and LAST_BIT must be a selected component.) Or does it include "subcomponents" as well?

One might argue that the use of "component" rather than "subcomponent" was an oversight. Moreover, consider:

```
type REC_INNER is record
    RI : INTEGER;
```

```
        end record;

        type ARR_REC is array (1..9) of REC_INNER;

        type REC_OUTER is record
            RX : INTEGER;
            AM : ARR_REC;
        end record;

        OBJ : REC_OUTER;
        INT : INTEGER;

        ...
        INT := OBJ.AM(INT).RI'FIRST_BIT;     -- Legal? (yes)
```

This would seem to be legal no matter how restrictive an interpretation one takes. But if an implementation can handle this, why should it not also handle the first example?

!response 86-07-07

For P'FIRST_BIT, P'LAST_BIT, and P'POSITION, A(16,23,34) state:

> For a prefix P that denotes a component of a record object ...

3.3(7) states:

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

Consequently, in the first example although OBJ.A is a component of the record object OBJ of type REC, OBJ.A(2) is not. In the second example, OBJ.AM(INT).RI is a component of the record object OBJ.AM(INT) of type REC_INNER. Although the implementation complexity of the two examples might be similar, the Standard is clear on the legality of the two cases.

Ada Commentary ai-00365-bi.wj downloaded on Thu May 12 09:50:02 EDT 1988

!standard 12.03 (17) 86-07-23 AI-00365/05
!class binding interpretation 85-07-21
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-20
!status work-item 86-01-24
!status received 85-07-21
!references 83-00583
!topic Actual parameter names are evaluated in generic instantiations

!summary 86-01-24

In a generic instantiation, the names appearing as actual parameters are evaluated.

!question 86-01-24

The elaboration of a generic instantiation is defined in 12.3(17). The paragraph specifies that

> Each expression supplied as an explicit generic actual parameter is first evaluated, as well as each expression that appears as a constitutent of a variable name or entry name supplied as an explicit generic actual parameter.

This wording does not state that the name of a variable associated with an in out parameter is actually evaluated, i.e., it does not say that the object denoted by such an actual parameter is determined. In particular, consider the variable PTR.all, which contains no expressions at all.

Clearly the intent is to require that all expressions be evaluated and that the names of all variables be evaluated, in an order that is not defined by the language. Has some wording been omitted?

!recommendation 86-01-24

Each name serving as an actual generic parameter is evaluated when a generic instantiation is elaborated.

!discussion 86-01-24

As the question notes, the current wording does not cover the evaluation of names such as PTR.all and C(I) when they serve as an actual in out parameter, but the intent was that the name of a variable serving as an actual in out parameter be evaluated to determine the entity denoted by the name.

Ada Commentary ai-00370-ra.wj downloaded on Thu May 12 09:50:03 EDT 1988

!standard 08.03 (16) 87-01-13 AI-00370/06
!class ramification 86-01-24
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-20
!status work-item 86-01-24
!status received 85-07-30
!references 83-00588
!topic Visibility of subprogam names within instantiations

!summary 86-02-27

Any declaration with the same designator as a subprogram instantiation is not visible, even by selection, within the instantiation.

!question 86-01-28

Consider the following example:

```
package PACKAGE_1 is
     generic
     procedure PROCEDURE_1;
end PACKAGE_1;

with PACKAGE_1;
package PACKAGE_2 is
     procedure PROCEDURE_1 is new PACKAGE_1.PROCEDURE_1;  -- * --
end PACKAGE_2;
```

According to 8.3(16), the starred line is illegal, since we are within a generic instantiation that declares a subprogram and therefore "every declaration with the same designator as the subprogram is hidden" and "where hidden in this manner, a declaration is visible neither by selection nor directly."

In view of the example, was 8.3(16) as written the intent? Or was the intent closer to the following?

> Where hidden in this manner, a declaration is also not visible by selection if its expanded names are the same as those containing the subprogram specification or generic instantiation.

(Expanded names, not name, because packages can be renamed.)

!response 86-03-04

8.3(16) says:

> Within the specification of a subprogram, every declaration with

the same designator as the subprogram is hidden; the same holds within a generic instantiation that declares a subprogram, and within an entry declaration or the formal part of an accept statement; where hidden in this manner, a declaration is visible neither by selection nor directly.

Consider the following example:

```
package PACK is
     function F return INTEGER;
     function F (I : INTEGER := PACK.F) return INTEGER;  -- illegal
end PACK;
```

By 8.3(16), the use of F in the default expression is illegal. Without this restriction, one is faced with the problem of how to determine whether a call like PACK.F denotes the containing function specification F before processing of that specification is completed, that is, before there is even a parameter and result type profile for F that can be used in overload resolution.

The restriction of 8.3(16) is needed also for generic subprogram instantiations because if the generic subprogram has a formal subprogram parameter and/or a formal 'in' parameter, then the corresponding actual parameters in an instantiation can cause a similar kind of overload resolution problem as illustrated above for a non-generic subprogram. For example:

```
generic
     J : INTEGER;
function GF (I : INTEGER := J) return INTEGER;

package PACK is
     function F return INTEGER;
     function F is new GF (PACK.F);      -- illegal
end PACK;
```

!appendix 87-01-13

****

!section 08.03 (16) Geoff Mendal 86-12-04 83-00870
!version 1983
!topic Assumption in AI-00370/05

The very last example in AI-00370/05 assumes that both declarations are inside the same declarative region, and are not each compilation units. This must be so since if each were compilation units, the use of "is new GF (PACK.F)" would be semantically invalid since no visibility to GF has been established. I would suggest placing these declarations in a declare block, or adding a context clause ("with GF") to package PACK.

Ada Commentary ai-00371-bi.wj downloaded on Thu May 12 10:45:02 EDT 1988

| !standard 13.01 (07) 86-07-23 AI-00371/05
| !standard 02.08 (09)
| !class binding interpretation 85-07-30
| !status approved by WG9/AJPO 86-07-22
| !status approved by Director, AJPO 86-07-22
| !status approved by WG9/Ada Board 86-07-22
| !status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-20
!status work-item 86-01-24
!status received 85-07-30
!references 83-00594
!topic Representation clauses containing forcing occurrences

!summary 86-01-24

An expression in a representation clause is illegal if it contains a forcing occurrence for the type whose representation is being specified.

!question 86-01-24

Is the occurrence of a type name within an expression of its own representation clause legal?

!recommendation 86-01-24

A forcing occurrence for a type is not allowed in an expression of a representation clause for the type.

!discussion 86-05-20

Consider the following examples of representation clauses:

```
type T is delta 1.0 range -10.0 .. 10.0;
for T'SMALL use T'SMALL/2;

type E is (EA, EB);
for E use (EA => E'POS(EA)-1, EB => E'POS(EB));
for E'SIZE use E'SIZE*2;
```

13.1(6) says a forcing occurrence for a type is

> any occurrence other than in ... a representation clause for the type itself. In any case, an occurrence within an expression is always forcing.

13.1(7) says:

> A representation clause for a given entity must not appear after an occurrence of the name of the entity if this occurrence forces a default determination of representation for the entity.

This wording does not clearly cover the case of a representation clause that contains a forcing occurrence for the type named in the clause, since such a clause does not occur "after" the forcing occurrence. However, it was clearly the intent to forbid such potentially circular references to a type's representation. Accordingly, all the representation clauses given above are illegal.

Ada Commentary ai-00376-ra.wj downloaded on Thu May 12 10:45:04 EDT 1988

!standard 04.05.05 (10) 86-12-01 AI-00376/04
!class ramification 85-08-22
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status committee-approved (9-0-0) 85-11-20
!status work-item 85-10-31
!status received 85-08-22
!reference AI-00020, 83-00603
!topic Universal real operands with fixed point * and /

!summary 85-10-31

An expression having type universal_real is not allowed as an operand of a fixed point multiplication or division operation. The possibility of adopting a more liberal rule in a future version of the language will be studied.

This commentary extends the conclusions of AI-0020 to cover all expressions of type universal_real, not just those having the form of a real literal.

!question 85-10-31

Multiplication and division are allowed for operands of any fixed point type, but the Standard appears to forbid fixed point multiplications and divisions involving expressions having the type universal_real. Is this an oversight?

!response 85-10-31

The expression

```
Fixed_Point_Type (V1 * UR)
```

is illegal when V1 has a fixed point type and UR has type universal_real (e.g., if UR is a named number) because * is not defined for one operand of type universal real and the other operand of some fixed point type; hence, the expression can only be legal if the real literal can be converted (implicitly) to a unique fixed point type [4.6(15)]. But there is no unique fixed point type to which UR can be converted. (There is no unique fixed point type since there are always at least two fixed point types whose scope includes every compilation unit -- the type DURATION and the anonymous predefined fixed point type specified in 3.5.9(7)). The same argument applies to fixed point division.

It would be possible to extend the language so * and / were defined for combinations of fixed point types and universal real values. However, performing such arithmetic operations with the accuracy required by the Standard is not straightforward. A literal such as 3.14 must be treated as an exact quantity, i.e., as though its 'SMALL were 3.14. (Then 3.14 will be represented exactly as a model number.) Consequently, to evaluate V1 * 3.14 correctly requires the same support from a compiler as that required when the compiler has decided to support 'SMALL values that are not powers of 2.

Since an implementation does not have to support the representation clause for 'SMALL (at least for values that are not powers of two; see 13.1(10)), it would be inconsistent to require such support in order to process V1 * 3.14.

The possibility of removing this restriction in a future version of the Standard will be studied, but the current Standard is unambiguous on this point.

This discussion extends the discussion and conclusion of AI-00020 to include all expressions having a universal_real type. AI-00020 covered just expressions having the form of a real literal.

Ada Commentary ai-00379-co.wj downloaded on Thu May 12 10:50:01 EDT 1988

!standard 13.05 (08) 86-12-01 AI-00379/03
!class confirmation 86-05-13
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (8-0-1) 86-05-13 (pending editorial review)
!status received 85-08-22
!references 83-00605
!topic Address clauses for entries of task types

!summary 86-07-03

If an interrupt is linked to an entry of more than one task object (of the same type), the program is erroneous.

!question 86-07-03

In the following example:

```
task type T is
    entry E;
    for E use at 16#DEF#;
end T;

OBJ1, OBJ2 : T;
```

Are we allowed to create more than one object of task type T?

!response 86-07-07

13.5(8) states:

> Address clauses should not be used to achieve overlays of objects or overlays of program units. Nor should a given interrupt be linked to more than one entry. Any program using address clauses to achieve such effects is erroneous.

Thus, a program that contains the above example would be erroneous, and its effects are unpredictable (i.e., implementation dependent; 1.6(7)). This interpretation holds also in the case of task objects of the same type declared in parallel blocks:

```
procedure P is
    task type T is
        entry E;
        for E use at 16#DEF#;
    end T;
begin
    BLK1 : declare
        OBJ1 : T;    -- OBJ1.E
    begin
```

```
            ...
        end BLK1;
        ...
        BLK2 : declare
            OBJ2 : T;     -- OBJ2.E
        begin
            ...
        end BLK2;
    end;
```

Execution of procedure P is erroneous if both blocks are executed.

Ada Commentary ai-00384-ra.wj downloaded on Thu May 12 10:50:02 EDT 1988

!standard 07.04.01 (04) 86-07-23 AI-00384/05
!class ramification 85-11-20
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9-0-0) 85-11-20
!status work-item 85-10-31
!status received 85-09-02
!references 83-00627
!topic Use of an incomplete private type in a formal type declaration

!summary 85-12-29

An incompletely declared private type cannot be used in the declaration of a generic formal type.

!question 86-01-20

Consider the following example:

```
package P is
    type PR is private;
    subtype S15 is INTEGER range 1..5;

    generic
        type FORMAL is array (S15) of PR;   -- illegal use of PR
    package ... end;
```

Is the use of PR in FORMAL's declaration illegal?

!response 85-12-29

7.4.1(4) says:

> Within the specification of the package that declares a private type and before the end of the corresponding full type declaration, a restriction applies to the use of a name that denotes the private type ... The only allowed occurrences of such a name are in a deferred constant declaration, a type or subtype declaration, ...

1.5(6) says:

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

The phrasing in 7.4.1(4) thus refers to the syntactic rule, type_declaration. Since a generic formal type is not declared, syntactically, by a type_declaration, an incompletely declared private type cannot be used in the declaration of a generic formal type.

Ada Commentary ai-00387-nb.wj downloaded on Thu May 12 11:45:02 EDT 1988

| !standard 11.01 (06) 87-02-23 AI-00387/05
!standard 03.05.04 (10)
!standard 03.05.06 (06)
!standard 04.05 (07)
!standard 04.05.05 (12)
!standard 04.05.07 (07)
!standard 04.10 (05)
!class non-binding interpretation 85-09-16
| !status approved by WG9/AJPO 87-02-20
| !status approved by Director, AJPO 87-02-20
| !status approved by Ada Board (21-0-0) 87-02-19
| !status approved by WG9 86-05-09
!status committee-approved (10-0-2) 86-02-20
!status committee-approved 85-09-04 (pending letter ballot)
!status received 85-09-16
!references AI-00115, AI-00159, AI-00311, AI-00312, AI-00368, 83-00628
!topic Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR

!summary 86-05-20

Wherever the Standard requires that NUMERIC_ERROR be raised (other than by a raise statement), CONSTRAINT_ERROR should be raised instead.

This interpretation is non-binding.

!question 85-10-31

The Standard requires that NUMERIC_ERROR be raised by the execution of predefined numeric operations, including cases "where an implementation uses a predefined numeric operation for the execution, evaluation, or elaboration of some construct." The implicit use of numeric operations when evaluating certain constructs means NUMERIC_ERROR can be raised under almost any circumstance. For example, questions have shown that NUMERIC_ERROR can be raised by the conversion of bounds in an array conversion (AI-00368). Other questions have arisen in which the Standard requires that CONSTRAINT_ERROR be raised although the most efficient implementation technique would cause NUMERIC_ERROR to be raised. For example, INTEGER'PRED(INTEGER'FIRST) must raise CONSTRAINT_ERROR, not NUMERIC_ERROR (see AI-00311), and evaluation of the following null string must also raise CONSTRAINT_ERROR instead of NUMERIC_ERROR, even though the upper bound might be computed as INTEGER'FIRST + length("") - 1:

```
type ARR is array (INTEGER range <>) of CHARACTER;
NS : constant ARR := "";                    -- CONSTRAINT_ERROR
```

In addition, checking a scalar value against a range constraint can sometimes be efficiently implemented by checking if the addition of a suitable constant value causes overflow, but the efficiency of this check is negated by the requirement that CONSTRAINT_ERROR be raised instead of NUMERIC_ERROR.

Is there really any benefit in distinguishing between NUMERIC_ERROR and CONSTRAINT_ERROR?

!recommendation 86-05-20

Where the Standard requires that NUMERIC_ERROR be raised (other than by a raise statement), CONSTRAINT_ERROR should be raised instead.

This recommendation is non-binding. It is anticipated that in the future, although NUMERIC_ERROR will remain as a predefined exception, no predefined operation will be required to raise it; CONSTRAINT_ERROR will be required instead.

!discussion 85-12-29

It is difficult for programmers to take full advantage of the distinction between NUMERIC_ERROR and CONSTRAINT_ERROR because either can be raised in many situations. For example, consider a numeric conversion:

```
X := NUMERIC_TYPE(Y);
```

What exception will be raised if the value of Y lies outside the range specified for X? NUMERIC_ERROR will be raised if Y's value (after conversion) lies outside the range of X's base type (3.5.4(10) and 4.5.7(7), but otherwise CONSTRAINT_ERROR will be raised. In general, a handler must mention both exceptions since it cannot be predicted in advance which exception will be raised.

In addition, the necessity to distinguish between the two exceptions can reduce the efficiency of code generated for the SUCC and PRED attributes (since these must raise CONSTRAINT_ERROR instead of NUMERIC_ERROR although the straightforward implementation technique is to use integer addition and subtraction to implement these operations; if overflow occurs, it must be interpreted as CONSTRAINT_ERROR, or else checks for the exception situation must be made before adding or subtracting).

The difficulty of making use of the distinction between these exceptions plus the fact that distinguishing between them has an adverse effect on code efficiency (particularly when performing range checks) indicates that the distinction between the exceptions is inappropriate. Implementations should be allowed to raise just one of these exceptions when a predefined operation is performed. Raising CONSTRAINT_ERROR in all cases is preferable to sometimes raising NUMERIC_ERROR.

Whenever programmers provide a handler for NUMERIC_ERROR, they should also provide a choice for CONSTRAINT_ERROR. Implementations are encouraged to warn users when they fail to do this.

Ada Commentary ai-00388-ra.wj downloaded on Thu May 12 11:45:04 EDT 1988

!standard 02.08 (04) 87-01-13 AI-00388/05
!class ramification 85-10-29
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status committee-approved (7-0-2) 85-11-22
!status work-item 85-10-29
!status received 85-09-16
!references 83-00298, 83-00635
!topic Pragmas are allowed in a generic formal part

!summary 85-10-10

Pragmas are allowed in a generic formal part.

!question 85-10-10

2.8(4) says:

> [Pragmas are allowed] after a semicolon delimiter, but not within a formal part or discriminant part.

In addition, 2.8(5) says:

> [Pragmas are allowed] at any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", ...

Since a generic_formal_part is different from a formal_part, and since a generic formal part consists of a sequence of generic_parameter_declarations, does this mean that pragmas ARE allowed in a generic formal part, or was it the intent to forbid pragmas in generic formal parts as well?

!response 86-01-07

As noted in the question, the Standard does allow pragmas to appear in generic formal parts. In particular,

```
generic
    pragma PAGE;
    X : INTEGER;
    pragma PAGE;
package P;
```

is legal.

Pragmas are terminated with semicolons and are only allowed in contexts where semicolons are used to terminate constructs. Although a generic formal part, a formal part, and a discriminant part have similar functions, they use the semicolon differently. In a generic formal part, the semicolon appears at the end of each declaration. In a formal part and discriminant part, the semicolon separates declarations. In particular, a semicolon does not follow

the last declaration. If a pragma were allowed after the last declaration, it would not be separated from the preceding declaration by a semicolon, since no semicolon is present. A special rule would be needed either to insert a semicolon in such a case or to forbid pragmas after the last declaration. Rather than provide a special rule, it was decided to disallow pragmas in contexts where semicolons separate constructs, namely in formal parts and discriminant parts.

!appendix 87-01-13

****

!section 02.08 (04) Geoff Mendal 86-12-05 83-00871
!version 1983
!topic Error in AI-00388/04

The example used in the "!response" section of AI-00388/04 is syntactically invalid, not due to pragmas, but rather due to a missing "is end" after "package P".

```
generic
  . . .
package P;
```

should be transformed to one of the following:

```
generic                    generic
  . . .                      . . .
package P is end;          procedure P;
```

Ada Commentary ai-00396-cr.wj downloaded on Thu May 12 11:50:01 EDT 1988

!standard 06.04.01 (09) 86-07-23 AI-00396/03
!class correction 85-03-07
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9-0-0) 85-11-20
!status work-item 85-10-09
!status received 85-03-07
!references AI-00025, 83-00510
!topic Correction to discussion of AI-00025

!summary 85-10-09

The discussion section of AI-00025/07 should be corrected. Instead of saying,

> "The effect of the call, CALL_Q_NOW.CALL_Q.Q(Y), is to assign an invalid value to P.Z"

the discussion should say,

> "The effect of elaborating CALL_Q_NOW is to assign an invalid value to P.Z".

!question 85-12-30

The discussion of AI-00025/07 contains the expanded name CALL_Q_NOW.CALL_Q.Q, but there is no Q declared in the instantiation for CALL_Q. The intention is merely to refer to the effect of the call Q(Z). Can this be done simply by referring to the effect of elaborating the instantiation for CALL_Q_NOW?

!recommendation 85-10-09

Correct the discussion of AI-00025 by replacing:

> "The effect of the call, CALL_Q_NOW.CALL_Q.Q(Y), is to assign an invalid value to P.Z"

with

> "The effect of elaborating CALL_Q_NOW is to assign an invalid value to P.Z".

!discussion 85-10-09

The discussion section of commentary AI-00025/07, as approved by the ADA Board and ISO WG9 (then known as WG14), contains an example which can be summarized as follows:

```
package P is ... end P;

generic
     Y : in out P.T;
package CALL is
end CALL;

package body CALL is
begin
     Q (Y);
end CALL;

package body P is
     Z : T range 0..1 := 0;
     package body PP is
          package CALL_Q is new CALL(Z);
     end PP;
end P;

package CALL_Q_NOW is new P.PP;
```

With respect to this example, the discussion says:

> The effect of the call, CALL_Q_NOW.CALL_Q.Q(Y), ...

The expanded name is incorrect because Q is not declared within CALL_Q and Y is the generic parameter of CALL. To see this, note that the generic instantiation first implicitly generates the following instance:

```
package body CALL_Q_NOW is
   package CALL_Q is new CALL(Z);
      -- bound to Z although Z is not visible
end CALL_Q_NOW;
```

and then the inner instance:

```
package body CALL_Q is
begin
   Q(Z);
end CALL_Q;
```

The elaboration of this instance now calls Q(Z) as expected.

The intention of the discussion is merely to refer to the effect of the call Q(Z). This can be done simply by saying:

> The effect of elaborating CALL_Q_NOW ...

Ada Commentary ai-00397-bi.wj downloaded on Thu May 12 11:50:02 EDT 1988

| !standard 04.08 (13) 86-07-23 AI-00397/04
| !standard 04.08 (05)
| !class binding interpretation 85-10-24
| !status approved by WG9/AJPO 86-07-22
| !status approved by Director, AJPO 86-07-22
| !status approved by WG9/Ada Board 86-07-22
| !status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9-0-0) 85-11-20
!status work-item 85-10-29
!status received 85-10-24
!reference AI-00150, 83-00675, 83-00223
!topic Checking the designated subtype for an allocator

!summary 85-12-29

When evaluating an allocator, a check is made that the designated object belongs to the allocator's designated subtype. CONSTRAINT_ERROR is raised if this check fails. This check can be made any time before evaluation of the allocator is complete. In particular, it is not defined whether this check is performed before creation of a designated object, evaluation of any default initialization expressions, or evaluation of any expressions contained in the allocator.

!question 85-10-29

Consider the following example:

```
type REC (D : INTEGER := FUNC_D; E : INTEGER := FUNC_E) is
    record
        C : INTEGER := FUNC_C;
    end record;

type AC_REC_3 is access REC(3, 3);

VAR1 : AC_REC_3 := new REC(4, 3);          -- Can FUNC_C be called?
VAR2 : AC_REC_3 := new REC'(IDENT_INT(4),
                            FUNC);         -- Must FUNC be called?
VAR3 : AC_REC_3 := new REC;                -- Can FUNC_C be called?

subtype INT_10 is INTEGER range 1..10;
type AC_INT_10 is access INT_10;

VAR4 : AC_INT_10 := new INTEGER'(11);      -- is an exception raised?
```

The current wording of 4.8 does not require any check to be made that the objects designated by these allocators belong to the designated subtype of the allocator. Note that in none of these cases will evaluation of the subtype indication, qualified expression, or initialization expression cause any exception to be raised. Assuming, however, that a check must be made, is it required that the check be made before an object is created or any initialization expressions are evaluated? Is it allowed to make the check as

soon as possible, e.g., before all discriminant expressions have been evaluated? Specifically,

1) For VAR1, can the initialization expression for component C be evaluated before CONSTRAINT_ERROR is raised?

2) For VAR2, can CONSTRAINT_ERROR be raised before evaluating all the discriminant expressions?

3) For VAR3, can the initialization expression for component C be evaluated before CONSTRAINT_ERROR is raised? Must both default discriminant values be evaluated before the constraint check is made?

4) Is CONSTRAINT_ERROR raised when VAR4's initialization expression is evaluated?

!recommendation 85-12-29

CONSTRAINT_ERROR is raised if the object created by an allocator does not belong to the designated subtype of the allocator. This check is performed when evaluating the allocator. (It is not further defined when the check is performed.)

!discussion 85-10-29

4.8(6) specifies the rules for evaluation of allocators:

> For the evaluation of an allocator, the elaboration of the subtype indication or the evaluation of the qualified expression is performed first. The new object is then created. Initializations are then performed as for a declared object (see 3.2.1); the initialization is considered explicit in the case of a qualified expression; any initializations are implicit in the case of a subtype indication. Finally, an access value that designates the created object is returned.

4.8(13) mentions the exceptions that can be raised by the evaluation of an allocator:

> The exception STORAGE_ERROR is raised by an allocator if there is not enough storage. Note also that the exception CONSTRAINT_ERROR can be raised by the evaluation of the qualified expression, by the elaboration of the subtype indication, or by the initialization.

Note that 4.8(13) does not specify any additional checks that are to be performed when evaluating an allocator, but clearly, when evaluating the allocators for VAR1, VAR2, VAR3, and VAR4, no exception need be raised when any subtype indication is elaborated or when any qualified expression or initialization expression is evaluated.

It is the intent of the Standard that objects designated by access values

always satisfy any constraint imposed on the access type, but it is not the intent to specify exactly when such a constraint must be checked. Consequently, although there is a required check that must be performed when an allocator is evaluated, it is not further specified when the check must be performed.

The specific answers to the questions, therefore, are:

1) for VAR1, the initialization expression can be evaluated before CONSTRAINT_ERROR is raised.

2) for VAR2, CONSTRAINT_ERROR can be raised before all discriminant expressions have been evaluated.

3) for VAR3, any or all of the default expressions can be evaluated before CONSTRAINT_ERROR is raised.

4) CONSTRAINT_ERROR is raised when the allocator in VAR4's initialization is evaluated.

(This commentary replaces AI-00150, which gave a narrower interpretation that is consistent with the interpretation stated here.)

Ada Commentary ai-00398-bi.wj downloaded on Thu May 12 12:45:01 EDT 1988

!standard 12.03 (05) 87-06-18 AI-00398/08
!standard 03.04 (11)
!class binding interpretation 85-10-24
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (22-0-0) 87-02-19
!status panel/committee-approved (9-0-0) 86-11-13 (by ballot)
!status panel/committee-approved (5-0-1) 86-09-11 (pending letter ballot)
!status work-item 85-10-29
!status received 85-10-24
!references AI-00367, AI-00409, 83-00422, 83-00589, 83-00680, 83-00681, 83-00682, 83-00694, 83-00735, 83-00816, 83-00852, 83-00853
!topic Operations declared for types declared in instances

!summary 87-02-23

If the parent type in a derived type definition is a generic formal type, the operations declared for the derived type in the template are determined by the class of the formal type. The operations declared for the derived type in the instance are determined by the type denoted by the formal parameter.

Similarly, if the component type of an array type is a generic formal type or if the designated type of an access type is a generic formal type, the operations declared for the array and access type in the template depend on the class of the formal type. If the array and access type declarations do not occur in the generic formal part, then the operations declared for these types in a generic instance are determined by the type denoted by the formal parameter in the instance.

If the designated type in an access type declaration is an incomplete type, additional operations can be declared for the access type by the full declaration of the incomplete type (7.4.2(7-8)). If the full declaration declares a type derived from a generic formal type, the additional operations (if any) declared for the access type in the template are determined by the class of the formal type. The additional operations declared for the access type in the instance are determined by the type denoted by the formal parameter.

Similar rules apply when the parent type, component type, or designated type is derived, directly or indirectly, from a generic formal type.

!question 86-07-31

Consider the following generic units and instantiations:

```
-- Example 1
generic
     type DISCRETE is (<>);
package GP1 is
```

```
        type AR is array (1..5) of DISCRETE;
        Y1 : AR := (1..5 => DISCRETE'VAL(0));
        Y2 : AR := Y1 and Y1;          -- illegal
        Y3 : AR := "abcde";            -- illegal
    end GP1;

    package P11 is new GP1 (BOOLEAN);
    package P12 is new GP1 (CHARACTER);
    use P11, P12;

    X1 : P11.AR := ...;
    X2 : P11.AR := X1 and X1;          -- legal? (yes)
    X3 : P12.AR := "abcde";            -- legal? (yes)
```

The initialization expression for GP1.Y2 is illegal since no logical operators are declared for GP1.AR; hence, no such operators are visible within the generic unit. Similarly, the string formation operation is not declared within the unit. Are such operations declared in the instances P11 and P12, respectively? If so, the initialization expressions for X2 and X3 are legal. If the initialization expressions are legal, note that different operations are implicitly declared in P11 and P12, and some operations declared in the instance are not declared in the template. Is this correct?

Similar questions arise when deriving from a generic formal type:

```
    -- Example 2
    generic
        type PRIV is private;
    package GP2 is
        type NT2 is new PRIV;
    end GP2;

    package R2 is
        type T2 is range 1..10;
        function F return T2;
    end R2;

    package P2 is new GP2 (R2.T2);
    use P2;

    XX1 : P2.NT2 := 5;                 -- legal? (yes)
    XX2 : P2.NT2 := XX1 + XX1;         -- legal? (yes)
    XX3 : P2.NT2 := P2.F;              -- legal? (yes)
```

The initialization expression for XX1 is legal if an implicit conversion from universal_integer to P2.NT2 is declared in P2. Similarly, the use of "+" in XX2's initialization is legal if the predefined "+" operator is declared for P2.NT2, and finally, the initialization expression for XX3 is legal if function F is derived within the instance. Are these initialization expressions legal?

!recommendation 87-02-23

If the parent type in a derived type definition is a generic formal type, the operations declared for the derived type in the template are determined by the declaration of the formal type. The operations declared for the derived type in the instance are determined by the actual type denoted by the formal parameter. Similarly, if the parent type in the template is a type derived directly or indirectly from a generic formal type, the operations declared for the derived type in the template are determined by the usual rules (see 3.4 and AI-00367). The operations declared for the derived type in the instance are determined by the operations declared for the parent type in the instance and by the actual type denoted by the formal parameter.

If the component type of an array type is a generic formal type or is a type derived directly or indirectly from a generic formal type, the operations declared for the array type in the template are determined by the declaration of the formal type. The operations declared for the array type in the instance are determined by the actual type denoted by the formal parameter.

If the designated type of an access type is a generic formal type or is a type derived directly or indirectly from a generic formal type, the operations declared for the access type in the template are determined by the declaration of the formal type. The operations declared for the access type in the instance are determined by the actual type denoted by the formal parameter.

If the designated type in an access type declaration is an incomplete type, additional operations can be declared for the access type by the full declaration of the incomplete type (7.4.2(7-8)). If the full declaration declares a type derived directly or indirectly from a generic formal type, the additional operations (if any) declared for the access type in the template are determined by the declaration of the formal type. The additional operations declared for the access type in the instance are determined by the actual type denoted by the formal parameter.

!discussion 86-09-28

12.3(5) says:

> The instance is a copy of the generic unit, apart from the generic formal part; thus the instance of a generic package is a package. ... For each occurrence, within the generic unit, of a name that denotes a given entity, the following list defines which entity is denoted by the corresponding occurrence within the instance.

In other words, an instance of a generic unit is essentially an ordinary unit, but with slightly different rules affecting the visibility of names. These rules (12.3(6-16)) change the interpretation of names within the instance so that, notwithstanding the usual interpretation of names in a unit, formal parameter names refer to corresponding actual parameters in the instance; names of locally declared entities refer to corresponding local

entities in the instance; and non-local names refer to the same non-local entities in the instance.

The wording of 12.3(5) suggests that, apart from specific changes regarding the interpretation of names occurring in both the template and the instance, the instance is considered to be an ordinary package or subprogram. In particular, the implicit declarations generated by type declarations in the instance are those that would be generated if the type declaration were written explicitly. An alternative view is that only the implicit declarations generated in the template are present in the instance (which, after all, is a "copy" of the template); no other declarations are created so no further analysis of the instance is necessary. These two views lead to different interpretations of the examples given in the question. In particular, if no new implicit declarations are created, the lines in the examples marked "legal?" will all be considered illegal, since the required operations were not declared in the template and will not be declared in the instance.

After considering both views, it seems that fewer anomalies arise if the recommended view is taken. In arriving at this conclusion, examples like the following have been considered:

```
-- Example 3
generic
     type T3 is range <>;
package GP3 is
     type NT3 is new T3;
     X : NT3 := 5;
     Y : NT3 := X + 3;        -- uses predefined "+" even in instances
end GP3;

package R3 is
     type S is range 1..10;
     function "+" (L, R : S) return S;
end R3;

package P3 is new GP3 (R3.S);
use P3;

Z : P3.NT3 := P3.X + 3;       -- uses redefined "+"
```

Within the template, the "+" in Y's initialization expression denotes the predefined "+" for NT3, so in the instance, P3, the initialization expression for P3.Y is evaluated using the predefined "+" for P3.NT3 (12.3(15)). The declaration of P3.NT3, however, derives a new "+" operation for P3.NT3. This derived subprogram hides the predefined "+", and so, in the initialization of Z, only the redefined "+" operation is visible. In the alternative view, no redefined "+" operation would be derived, so the initialization expression for Z would use the predefined "+" operation. For this example, the consequences of either view seem equally preferable. (To ensure the "+" in the initialization of P3.X is the user-defined "+", the generic unit should be supplied with a formal function parameter for "+" and an explicit "+"

function should be declared for type GP3.NT3; the explicit function should then be defined to call the formal "+" function.)

Now consider this example:

```
-- Example 4
generic
     type T4 is limited private;
package GP4 is
     type NT4 is new T4;
     X : NT4;
end GP4;

package P4 is new GP4 (BOOLEAN);

...

if P4.X then                          -- legal? (yes)
```

The issue here is whether P4.X has a boolean type, and hence is allowed in the condition of an if statement. If we take the recommended view, the literals TRUE and FALSE are derived for P4.NT4 as well as the other operations declared for a derived boolean type, and NT4 is clearly a boolean type. If we take the alternative view, no assignment operation is declared for NT4, there are no literals, and it is unclear whether P4.NT4 is a boolean type. This consequence of the alternative view is likely to seem peculiar to Ada programmers.

Now consider this example:

```
-- Example 5
generic
     type T5 (D : POSITIVE) is private;
package GP5 is
     type NT5 is new T5;
     X : NT5 (D => 5);
     Y : POSITIVE := X.D;
end GP5;

type REC (A : POSITIVE) is
     record
          D : POSITIVE := 7;
     end record;

package P5 is new GP5 (REC);

W1 : POSITIVE := P5.X.D;              -- value is 7
W2 : POSITIVE := P5.X.A;              -- value is 5
W3 : POSITIVE := P5.Y;                -- value is 5
```

Within the template, X.D refers to the discriminant of NT5, so within the instance, the initialization expression for P5.Y still uses the component

selection operation for the discriminant (12.3(15)). With the recommended view, the component selection operations declared for P5.NT5 are those that exist for the parent type, T5, which denotes REC (12.3(9)). Hence, P5.X.A denotes the discriminant of P5.X, and P5.X.D denotes the component. With the alternative view, the component selection operation of the template is copied to the instance, and P5.X.D denotes the discriminant of P5.X; P5.X.A is illegal because no such component selection operation is declared in the instance. Once again, the recommended view yields an intuitively satisfying result, while the alternative view produces subtle and confusing consequences.

A similar example can be given for access types:

```
-- Example 5a
generic
     type T5A (D : POSITIVE) is private;
package GP5A is
     type NT5A is access T5A;
     X : NT5A := new T5A (D => 5);
     Y : POSITIVE := X.D;
end GP5A;

type REC (A : POSITIVE) is
     record
          D : POSITIVE := 7;
     end record;

package P5A is new GP5A (REC);

W1 : POSITIVE := P5A.X.D;          -- value is 7
W2 : POSITIVE := P5A.X.A;          -- value is 5
W3 : POSITIVE := P5A.Y;            -- value is 5
```

Next consider:

```
-- Example 6
generic
     type T6 is private;
package GP6 is
     type NT6 is new T6;
end GP6;

package R6 is
     type ENUM is (ALPHA, BETA);
     procedure P (X : ENUM);
     package P6 is new GP6 (ENUM);
end R6;
```

With the recommended view, the enumeration literals for ENUM are derived for P6.NT6 and declared in P6. The procedure P is not derived within P6 because P is not derivable until the end of R6's visible part (3.4(11)).

Now suppose package P6 is inside a generic package:

```
-- Example 7
package R7 is
     type ENUM is (ALPHA, BETA);
     procedure P (X : ENUM);

     generic
     package GP7 is
          package P6 is new GP6 (ENUM);
     end GP7;
end R7;

package P7 is new R7.GP7;
```

Since the instance, P7, is outside R7's visible part, procedure R7.P is derivable and is declared as P7.P6.P. In the alternative view, since P is not declared in the instance, GP7.P6, it is also not declared in the instance P7.P6.

As a final example, the declaration of a derived type can sometimes require additional declarations to be declared for other types as well. For example:

```
-- Example 8
generic
     type T8 is private;
package GP8 is
     type INC;
     type ACC_INC is access INC;
     OBJ : ACC_INC;
     type INC is new T8;          -- additional ops for ACC_INC
end GP8;
```

In general the full declaration of INC can cause additional operations to be declared for ACC_INC (7.4.2(7-8)). For example, if T8 were STRING then it would be legal to write OBJ'FIRST or OBJ(1). The additional operations declared for ACC_INC in an instance of GP8 therefore depend on what type T8 denotes in the instance. For example:

```
package P8A is new GP8 (STRING);
package P8B is new GP8 (REC);    -- REC is declared in Example 5
```

With the recommended view, P8A.OBJ'FIRST is legal, and so is P8B.OBJ.A, since these attribute and component selection operations are declared in the instance for P8A.ACC_INC and P8B.ACC_INC, respectively. With the alternative view, since no new operations are declared in the instances, P8A.OBJ'FIRST and P8B.OBJ.A would be illegal.

To summarize, in considering Examples 1-8, the recommended view produces results that are readily understood by considering what the effects would be if an instance were written explicitly. The alternative view sometimes produces results that are subtle and confusing, and so has been rejected.

Ada Commentary ai-00405-ra.wj downloaded on Thu May 12 12:45:03 EDT 1988

!standard 04.10 (04) 86-12-01 AI-00405/06
!class ramification 86-05-13
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (9-0-0) 86-05-13 (pending editorial review)
!status work-item 86-01-24
!status received 85-12-04
!references AI-00103, 83-00659, 83-00665, 83-00736
!topic One nonstatic operand for a universal real relation

!summary 86-06-09

If the operands of a relational operator or membership test have the type universal_real and one or more of the operands is nonstatic, the static operands must be evaluated exactly. Doing so, however, does not impose a run time overhead.

!question 86-03-24

4.10(4) says that "if a universal expression is a static expression, then the evaluation must be exact." Consider a relational expression

NS = S

where NS is a nonstatic universal real expression and S is a static universal real expression. 4.10(4) allows NS to be evaluated using the most accurate floating point type supported by an implementation. Suppose the result of evaluating NS lies in the model interval M1 .. M2 and suppose S lies just outside this model interval. Since S has type universal_real, its exact value is a model number, so the rules in 4.5.7(10) require that NS = S return FALSE. To return the correct result for "=", S must, in general, be represented with arbitrary precision at run time, but imposing such a requirement on an implementation does not seem consistent with the idea that NS can be evaluated inexactly.

Must a static operand of a such a relational expression be evaluated exactly if the other operand is nonstatic? (A similar question can be posed for membership tests.)

!response 86-06-09

There is both a trivial and a less trivial resolution of this problem. Trivially 4.10(4) refers to "expressions," and in NS = S, for example, NS and S are not expressions according to the Ada syntax, but rather are simple_expressions, and possibly terms, factors, or primaries. Since the only expression involved is NS = S, which is nonstatic, infinite precision is not required anywhere, by this argument.

However, the intent of the Standard was that the word "expression" in these contexts should not be interpreted in the strict syntactic sense, but should

be interpreted to refer also to simple_expressions, relations, terms, factors, and primaries. Furthermore, in such constructs as

```
2.0 * PI * X,
```

the phrase 2.0 * PI is also to be considered an expression for the purposes of sections 4.9 and 4.10, even though it is not a term, but only a portion of a term.

Given that established Ada usage does not support the trivial resolution, there is also a non-trivial resolution that requires no compromise of the semantics. Suppose NS is a nonstatic universal real expression, and S is a static universal real expression. Consider the following relation.

```
NS relation S
```

The Standard (4.10(4)) indicates that S must be evaluated exactly. It also indicates (4.5.7(10)) that because the values of S and NS, being of type universal_real, are both model numbers, the relation itself must be evaluated exactly using the computed values of S and NS. It might at first seem that the expression S must be carried to full precision (i.e., as a ratio of arbitrarily large integers) at run time. This is not, in fact, the case.

Let LONG be the base type for the highest precision floating point numbers used by a given implementation. By abuse of notation, we shall also use it to denote the set of all values of type LONG. Let CEIL(x,LONG) be the least upper bound of the subset of LONG greater than or equal to x. Let FLOOR(x,LONG) be the greatest lower bound of the subset of LONG less than or equal to x. These may be undefined beyond the extrema of LONG. There are two cases, and the one that applies can be determined at compilation time.

1. The value of S is a member of LONG. In this case, the implementation is obvious.

2. S is not a member of LONG. Convert the relational expression according to the following table.

Expression	Transformed expression
NS > S, NS >= S, S < NS, S <= NS	NS > FLOOR(S,LONG), if the latter is defined or TRUE otherwise
NS < S, NS <= S S > NS, S >= NS	NS < CEIL(S,LONG), if the latter is defined or TRUE otherwise
NS = S, S = NS	FALSE
NS /= S, S /= NS	TRUE

We thus reduce everything to at worst the case of comparing a static member of LONG to a nonstatic member of LONG. As a consequence, it is possible to maintain the convenient "semantic fiction" that S is carried to infinite precision in the comparison without run time cost.

Ada Commentary ai-00406-bi.wj downloaded on Thu May 12 12:50:01 EDT 1988

!standard 03.09 (05) 87-08-06 AI-00406/05
!class binding interpretation 85-12-29
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (5-0-3) 86-11-13 (pending editorial review)
!status work-item 86-10-10
!status received 85-12-29
!references 83-00690
!topic Evaluating parameters of a call before raising PROGRAM_ERROR

!summary 87-01-19

It is not defined whether the check that the body of a subprogram has been elaborated is made before or after the actual parameters of a call have been evaluated.

Similarly, it is not defined whether the check that the body of a generic unit has been elaborated is made before or after the generic actual parameters of an instantiation have been evaluated.

!question 87-01-19

Is the check that the body of a subprogram has been elaborated made before or after the parameters of a call have been evaluated?

The corresponding question arises for generic instantiations.

!recommendation 86-12-03

The check that the body of a subprogram has been elaborated and the evaluation of each of the actual parameters of a call is made in some order that is not defined by the language.

Similarly, the check that the body of a generic unit has been elaborated and the evaluation of each of the generic actual parameters of an instantiation is made in some order not defined by the language.

!discussion 86-10-14

3.9(5) states:

> For a subprogram call, a check is made that the body of the subprogram is already elaborated.

6.4(1) states:

> [A subprogram call] invokes the execution of the corresponding subprogram body. The call specifies the association of the actual parameters, if any, with formal parameters of the

subprogram.

6.4.1(2) states:

> [An actual parameter associated with a formal parameter of mode in] is evaluated before the call.

6.4.1(4) states:

> The variable name given for an actual parameter of mode in out or out is evaluated before the call.

The Standard does not specify whether the check required by 3.9(5) is performed before or after or in conjunction with the evaluation of actual parameters. The intent was that these actions be performed in some order that is not defined by the language.

A similar analysis applies to generic instantiations:

3.9(7) states:

> For the instantiation of a generic unit that has a body, a check is made that this body is already elaborated.

12.3(17) states:

> For the elaboration of a generic instantiation, each expression supplied as an explicit generic actual parameter is first evaluated, as well as each expression that appears as a constituent of a variable name or entry name supplied as an explicit generic actual parameter; these evaluations proceed in some order that is not defined by the language. Then, for each omitted generic association (if any), the corresponding default expression or default name is evaluated; such evaluations are performed in the order of the generic parameter declarations. Finally, the implicitly generated instance is elaborated. The elaboration of a generic instantiation may also involve certain constraint checks ...

The Standard does not specify whether the check required by 3.9(7) is performed before or after or in conjunction with the evaluation of generic actual parameters. The intent was also that these actions be performed in some order that is not defined by the language.

Ada Commentary ai-00408-bi.wj downloaded on Thu May 12 12:50:02 EDT 1988

| !standard 10.03 (06) 87-08-20 AI-00408/11
| !class binding interpretation 86-05-13
| !status approved by WG9/AJPO 87-07-30
| !status approved by Director, AJPO 87-07-30
| !status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (7+3-1-1) 86-11 (by ballot; pending editorial review)
!status panel/committee-approved (6-2-1) 86-09-10 (pending letter ballot)
!status work-item 86-09-10
!status failed letter ballot (0-11-2) 86-09
!status committee-approved (5-1-4) 86-05-13 (pending letter ballot)
!status work-item 86-01-23
!references AI-00506, AI-00257, 83-00382, 83-00632, 83-00658, 83-00725, 83-00737, 83-00785
!topic Effect of compiling generic unit bodies separately

!summary 86-09-20

An implementation is allowed to create a dependence on a generic unit body such that successfully compiling (or recompiling) the body separately makes previously compiled units obsolete if they contain an instantiation of the generic unit. A similar dependence can be created for separately compiled subunits of a generic unit.

!question 86-01-23

If the body of a generic unit is recompiled, must all instantiations of the unit also be recompiled?

!recommendation 86-12-03

If a unit contains a generic instantiation, an implementation can create a dependence of the given unit on the generic body; if such a dependence is created and the body is compiled successfully in a separate compilation, the compiler must recognize this dependence when deciding on the need for recompilation. (See AI-00506 if the body is not compiled separately.) A similar dependence can be created and recognized for a subunit of a generic unit body when the subunit is given in a compilation separately from the parent unit or separately from a unit containing an instance.

| !discussion 87-08-20

Paragraphs 10.3(5-6) say:

> "... A compilation unit is potentially affected by a change in any library unit named by its context clause. A secondary unit is potentially affected by a change in the corresponding library unit. The subunits of a parent compilation unit are potentially affected by a change of the parent compilation unit. If a compilation unit is successfully recompiled, the compilation

units potentially affected by this change are obsolete and must be recompiled unless they are no longer needed. An implementation may be able to reduce the compilation costs if it can deduce that some of the potentially affected units are not actually affected by the change.

The subunits of a unit can be recompiled without affecting the unit itself. Similarly, changes in a subprogram or package body do not affect other units (apart from subunits of the body) since these compilation units only have access to the subprogram or package specification. An implementation is only allowed to deviate from this rule ... as described below."

Now consider the following set of files containing Ada compilation units:

```
----- File G_SPEC -----

    generic
        type T is private;
    package G is end;

----- File G_BODY -----

    package body G is
        X : T;
    end;

----- File TRY_G -----

    with G;
    procedure PROC is

        package MY_G is new G(INTEGER);

    begin
        ...
    end;

----------------------
```

Suppose we compile these files in the following order:

1. Compile file G_SPEC containing generic declaration G
2. Compile file G_BODY containing generic template G
3. Compile file TRY_G containing an instantiation of G
4. Compile file G_BODY again (whether edited and changed or not is irrelevant)

What is the status of unit PROC in the compilation library? Is PROC obsolete so that it must be recompiled before it can be linked and executed? Or is it perfectly okay "as is" and immediately ready to be linked and executed? As discussed below, the Standard allows PROC to be linked and executed without

being recompiled; it also allows an implementation to reject the attempt to recompile G_BODY.

According to the definitions given in 10.3(5, 6), the recompilation of G_BODY does not potentially affect PROC, so recompilation of PROC is not required. In particular, 10.3(6) notes that "changes to a subprogram or package body do not affect other compilation units." This wording applies to both generic and nongeneric bodies since "package body" and "subprogram body" are syntactic terms, and these syntactic constructs are used for generic as well as nongeneric units.

Paragraphs 10.3(7-8) allow certain deviations from the rules stated in 10.3(5-6) but these allowed deviations are not relevant to the example. 10.3(7) deals with pragma INLINE; AI-00200 allows additional dependences on generic unit bodies when pragma INLINE is applied to a generic unit or subprogram instantiation, but neither this paragraph nor this AI applies to the example, since pragma INLINE is not used. Paragraph 10.3(8) allows additional dependences to be created among the files belonging to a single compilation. This paragraph does not apply when generic unit bodies are processed in separate compilations, as in the example.

Paragraph 10.3(9) allows an implementation to require that a generic unit body (and its subunits, if any) be placed in the same compilation as its declaration. AI-00257 says, in effect, that an implementation is free to decide when to invoke the restriction allowed by 10.3(9). In particular, an implementation might decide to accept a separate compilation containing a generic unit body only if the generic unit has never been instantiated. Thus the attempt to compile G_BODY in step 2 of the example could succeed while the attempt to recompile G_BODY in step 4 could be rejected. If the attempt to recompile the body is rejected, a user can combine the specification and body in a single compilation, which then MUST be accepted (see AI-00506 for further discussion). Of course, recompiling the generic unit specification will make obsolete any unit that contains an instantiation of the generic unit -- the net effect of forcing recompilation of the generic specification is essentially the same as if the units containing the instantiations had been made obsolete directly as a result of recompiling the generic unit body. (The net effect, however, would be worse if the generic specification were given in a library package. Recompiling the library package (together with the generic unit body) would make obsolete all units that depend on the library package, whether or not they contain any generic instantiations. Thus, some unnecessary recompilation may be required.)

An implementation need not, in general, require that generic declarations and bodies (or subunits) be given in the same compilation. For example, after rejecting the recompilation of G_BODY, an implementation might accept the separate recompilation of just G_SPEC, thereby making obsolete all units that contain instantiations of G_SPEC. A user could then separately compile G_BODY, since there are no previous instantiations of unit G in the library. After recompiling G_BODY, the units containing the instantiations of G could then be compiled. So although an implementation can invoke 10.3(9) to reject certain separate recompilations of a generic unit body, it need not in practice require that generic specifications and bodies always be given in the same compilation.

In short, an implementation can require a sequence of recompilations that have the effect of making obsolete all units that contain instantiations of some generic unit. There seems to be little or no benefit to enforcing the notion that a unit containing a generic instantiation is unaffected when the corresponding generic unit body is recompiled. For all practical purposes, 10.3(9) and AI-00257 already allow an implementation to require recompilation of such instantiating units; it seems reasonable for recompilation to be required directly, rather than via the indirect ways allowed by 10.3(9) and AI-00257. Allowing an implementation to require such recompilations means separate compilation of generic unit bodies can be supported efficiently, and such support is considered highly desirable.

The recommendation does not ensure that separate compilation of generic specifications and bodies will be supported in every case, however. Consider the following example:

```
--- File STUB
procedure P is
     generic
          type T is private;
     procedure H;

     procedure H is separate;            -- stub for H

     procedure MY_H is new H(STRING);

begin null; end P;

--- File H_BODY
separate (P)
procedure H is
begin null; end H;
```

Suppose H_BODY is compiled separately. The recommendation allows an implementation to make procedure P obsolete when H_BODY is compiled, since P contains an instantiation of H. But recompiling P makes subunit H obsolete, since P is the parent of subunit H. Thus, there is no sequence of compilations that allows P and H_BODY to be compiled separately. If an implementation supports the recommendation given in this Commentary, such a pair of units can only be compiled if they are given together in a single compilation (see AI-00506 for further discussion).

Ada Commentary ai-00409-bi.wj downloaded on Thu May 12 13:45:03 EDT 1988

!standard 12.03 (05) 87-09-12 AI-00409/05
!standard 04.09 (11)
!class binding interpretation 85-10-24
!status approved by WG9/AJPO 87-07-30 (corrected in accordance with AI-00483)
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved 86-02-21 (after review)
!status committee-approved 85-11-20 (8-0-0) (pending editorial review)
!references AI-00398, AI-00483, 83-00821
!topic Static subtype names created by instantiation

!summary 86-03-05

A subtype can be nonstatic in a generic template and static in a corresponding instance.

!question 87-09-09

Consider the following generic unit and instantiation:

```
generic
    type T is (<>);
package SET_OF is
    type SET is array (T) of BOOLEAN;
end SET_OF;

package CHARACTER_SET is new SET_OF (CHARACTER);
subtype CHAR_SET is CHARACTER_SET.SET;
```

Is the index constraint for CHAR_SET static because the actual parameter corresponding to T is static? In particular, is the use of OTHERS in the following aggregate legal because the corresponding index subtype is static?

```
SET_OF_DIGITS : CHAR_SET := CHAR_SET'('0' .. '9' => TRUE,
                                      others => FALSE);
```

Note that the index subtype for SET_OF.SET is not static within the generic unit itself because T is a formal generic type.

!recommendation 86-03-05

For a generic instantiation, if an actual generic parameter is a static subtype, then every use of the corresponding formal parameter within the instance is considered to denote a static subtype, even though the formal parameter does not denote a static subtype in the generic template.

!discussion 86-03-05

4.9(11) specifies that a subtype is static if it is "a scalar base type,

other than a generic formal type; ..." Furthermore, 4.9(11) defines a static index constraint to be "an index constraint for which each index subtype of the corresponding array type is static, and in which each discrete range is static." Finally, a discrete range is static if it is a static subtype.

From these definitions, it is clear that within the generic TEMPLATE, SET does not have a static index type. But the question concerns SET's status in an instance.

12.3(5) says that a generic instantiation creates "a copy of the generic unit, apart from the generic formal part." Within this copy, a name that denotes a generic formal type in the template is considered to denote the corresponding subtype named by the associated generic actual parameter [12.3(9)]. So within the instance, it is clear that SET's index subtype is the type CHARACTER. Since CHARACTER is a static subtype, SET is declared with a static index constraint in the instance.

This view is consistent with the idea that the interpretation of names declared in an instance is determined by the usual rules of the language, after taking into consideration the association of generic formal parameters with their corresponding actual parameters.

Ada Commentary ai-00418-ra.wj downloaded on Thu May 12 13:45:06 EDT 1988

!standard 10.01 (03) 87-08-06 AI-00418/06
!class ramification 86-08-01
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-08-01
!status received 86-04-15
!references AI-00113, 83-00730, 83-00732
!topic Self-referencing with clauses

!summary 86-08-01

Circular dependences among library units are not allowed, i.e., the library unit being compiled cannot have the same name as a previously compiled library unit if a with clause for the unit being compiled establishes a direct or indirect dependence on the previously compiled unit.

!question 86-04-15

Consider the following sequence of compilations:

```
package REDECL1 is end;          -- Unit 1: OK

with REDECL1;
package REDECL1 is end;          -- Unit 2: error? (yes)

with REDECL1;
package REDECL2 is end;          -- Unit 3: OK

with REDECL2;
package REDECL1 is end;          -- Unit 4: error? (yes)
```

M. the attempt to compile units 2 and 4 be rejected? Why?

!response 86-09-28

Consider the following examples of self-referencing with clauses:

```
with P1;                         |   with P2;
package P1 is ... end P1;        |   procedure P2;
                                 |
with P3;                         |   with P4;
generic                          |   generic
package P3 is ... end P3;        |   procedure P4;
                                 |
with P5, GPKG;                   |   with P6, GPROC;
package P5 is new GPKG;          |   procedure P6 is new GPROC;
```

All these compilation units are illegal. 8.6(2) says:

> the declaration of every library unit is assumed to occur immediately within [the package STANDARD]. The implicit declarations of library units are assumed to be ordered in such a way that the scope of a given library unit includes any compilation unit that mentions the given library unit in a with clause.

The with clause must name a library unit, so the clause WITH Pn is illegal if there is no library unit Pn. If there is a library unit named Pn, then the unit being compiled cannot also have the name Pn if the unit being compiled is to be considered a library unit, since library unit names must be distinct [10.1(3)]. This reasoning also covers indirect circularities:

```
package P is ... end P;

with P;
package INDIRECT is ... end INDIRECT;

with INDIRECT;
package P is ... end P;    -- illegal
```

The attempt to compile the second package specification P cannot be successful because it requires the presence of library unit INDIRECT, which in turn, requires the presence of library unit P, thus violating the rule prohibiting identically named library units. Note the difference if the with clause is omitted:

```
package P is ... end P;
```

This recompilation of P can be successful because no with clause implies that any library unit called P must already exist. If P is successfully compiled, it is entered in the library (10.4(2)) and replaces any previously existing library unit called P (10.3(5)).

Also note the difference from an example like the following:

```
with P1;
package body P1 is ... end P1;
```

The with clause only requires that a library unit named P1 exist. Since the package body itself cannot be a library unit, the with clause is not trying to name a library unit that is being compiled, and so is allowed; unit P1 is legal as long as package specification P1 exists. 8.6(2) says P1's specification must be implicitly declared in STANDARD prior to the unit being compiled, and there is no difficulty in doing so. Compilation of P1 is unsuccessful if no library unit P1 exists or if the library unit P1 is not a package specification.

Now consider a similar situation for a subprogram body:

```
with P2;
procedure P2 is ... end P2;
```

If there is no library unit named P2, the with clause is illegal. If a library unit called P2 exists, the context clause is legal, but the procedure body is allowed only if it is acceptable as a conforming secondary unit. The procedure body will not be considered a secondary unit if the previously compiled library unit P2 is a package specification, a generic package specification, a generic instantiation, a function declaration, a generic function declaration, or a procedure body itself (as opposed to a procedure declaration; see AI-00225). If the previously compiled library unit is any of these kinds of unit, procedure body P2 must be considered as a library unit itself. Since the context clause requires that there already be a library unit named P2, and 10.1(3) requires that names of library units be distinct, the unit being compiled cannot both be a library unit and have the name P2. Hence, the above compilation unit is legal if and only if there exists a previously compiled conforming library unit declaration of P2 as a generic or non-generic procedure.

Ada Commentary ai-00425-co.wj downloaded on Thu May 12 13:50:01 EDT 1988

!standard 02.08 (08) 87-08-06 AI-00425/05
!class confirmation 86-06-19
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-10
!status received 86-06-19
!references 83-00743
!topic Restrictions on arguments of implementation-defined pragmas

!summary 87-01-19

An implementation can impose restrictions on the arguments of its implementation-defined pragmas (e.g., it can require that an argument be a static expression).

!question 86-10-10

May an implementation restrict expression arguments in implementation-defined pragmas to be static expressions?

!response 86-12-16

2.8(9) states that there may be restrictions imposed on the arguments of a pragma:

> A pragma (whether language-defined or implementation-defined) has no effect if its placement or its arguments do not correspond to what is allowed for the pragma.

2.8(8) states that the restrictions imposed on the implementation-defined pragmas must be described in Appendix F:

> An implementation may provide implementation-defined pragmas, which must then be described in Appendix F.

Thus, restrictions can be imposed by an implementation on the arguments of its implementation-defined pragmas.

Ada Commentary ai-00426-bi.wj downloaded on Thu May 12 13:50:02 EDT 1988

| !standard 03.02.01 (18) 87-06-18 AI-00426/05
| !standard 04.05.01 (03)
| !class binding interpretation 86-06-19
| !status approved by WG9/AJPO 87-06-17
| !status approved by Director, AJPO 87-06-17
| !status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (8-1-0) 86-09-10 (pending editorial review)
!status work-item 86-08-12
!status received 86-06-19
!references 83-00747
!topic Operations on undefined array values

!summary 86-09-15

If both operands of a predefined logical operator do not have the same number of components, CONSTRAINT_ERROR is raised, even if one of the operands has a scalar component with an undefined value.

!question 86-08-12

Consider the following example:

```
procedure MAIN is
    type ARR_TYPE is array (INTEGER range <>) of BOOLEAN;
    type R (D : INTEGER) is
        record
            A : ARR_TYPE (1 .. D);
        end record;
    R1 : R(1);
    R2 : R(2);
begin
    R2.A := R1.A or R2.A;     -- Erroneous? (no)
end MAIN;
```

Is execution of the program erroneous?

!recommendation 86-09-15

If both operands of a predefined logical operator do not have the same number of components, the execution of a program is not erroneous (CONSTRAINT_ERROR is raised).

!discussion 86-08-12

3.2.1(18) states:

> The execution of a program is erroneous if it attempts to apply a predefined operator to a variable that ha' a scalar subcomponent with an undefined value.

This seems to indicate that execution of the above program is erroneous, but 4.5.1(3) states:

> The operations on arrays are performed on a component-by-component basis on matching components, if any. ... A check is made that for each component of the left operand there is a matching component of the right operand, and vice versa. The exception CONSTRAINT_ERROR is raised if this check fails.

4.5.2(7) states:

> For ... two one-dimensional arrays of the same type, matching components are those (if any) whose index values match in the following sense: the lower bounds of the index ranges are defined to match, and the successors of matching indices are defined to match.

Since in the example, the operands of the OR operator do not have matching components, the intent is that no component value be accessed, and so CONSTRAINT_ERROR should be raised; execution of the program is not considered to be erroneous.

Ada Commentary ai-00444-ra.wj downloaded on Thu May 12 14:45:02 EDT 1988

!standard 09.07.02 (01) 87-08-06 AI-00444/05
!class ramification 86-07-10
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (9-0-1) 86-11-13 (pending editorial review)
!status work-item 86-08-12
!status received 86-07-10
!references 83-00768, 83-00865
!topic Conditional entry calls can be queued momentarily

!summary 86-12-15

A conditional entry call may (momentarily) increase the COUNT attribute of an entry, even if the conditional call is not accepted.

!question 87-06-04

9.7.2(1) says:

> A conditional entry call issues an entry call that is then canceled if a rendezvous is not immediately possible.

Can a conditional entry call affect the value of the COUNT attribute for the called entry, i.e., can the value of COUNT be increased after the call has been issued but before it is canceled?

!response 86-12-17

9.9(6) states:

> [The attribute COUNT for an entry of a task unit] yields the number of entry calls presently queued on the entry ...

Queueing is mentioned in 9.5(15):

> If several tasks call the same entry before a corresponding accept statement is reached, the calls are queued; there is one queue associated with each entry. Each execution of an accept statement removes one call from the queue.

This wording explains the effect when an entry call is issued, and applies to any entry call, e.g., if a timed entry call is issued (with a positive delay) and the called task has not reached a corresponding accept statement, the entry call is queued and the attribute COUNT for that entry is increased by one.

For conditional entry calls, if the called task has not reached a corresponding accept statement, the call is similarly queued (and, consequently, the attribute COUNT for that entry is increased by one), as

specified by 9.5(15), but then, 9.7.2(4) specifies that such a call is canceled (causing the attribute COUNT to be decreased by one). Whether the queueing of the call (i.e., the momentary increase of the COUNT attribute) is detectable in the called task depends on whether the actions of issuing and canceling the entry call are considered independent actions within the called task.

For a normal entry call, if a called task has reached an accept statement that has a queued call, the queued call is accepted. If the queue is empty, the call is accepted without being queued, and this holds for a conditional entry call as well.

If the called task is at a select statement, it can wait to decide which calls to accept. A conditional entry call could be queued while the called task is deciding whether to accept the call.

In short, a conditional entry call is queued, if necessary, until it is canceled and may (momentarily) increase the COUNT attribute of the entry.

Ada Commentary ai-00446-bi.wj downloaded on Thu May 12 14:45:03 EDT 1988

!standard 09.10 (06) 87-09-25 AI-00446/05
!standard 11.04 (01)
!class binding interpretation 86-07-10
!status approved by WG9/AJPO 87-07-29
!status approved by Director, AJPO 87-07-29
!status approved by Ada Board 87-07-29
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-4) 86-11-13 (pending editorial review)
!status work-item 86-08-12
!status received 86-07-10
!references 83-00772
!topic Raising an exception in an abnormally completed task

!summary 86-12-18

An exception can be propagated to an abnormally completed task that is engaged in a rendezvous or that is waiting for a task to be activated. If this occurs, the exception has no effect.

!question 87-09-25

11.4(1) says:

> When an exception is raised, normal program execution is abandoned and control is transferred to an exception handler.

This wording contains an implicit assumption that an exception can only be raised in a task that has not completed its execution, but the Standard allows an exception to be raised in a task that has completed its execution abnormally. To see how this can occur, consider the following examples. First consider a task that calls an entry and becomes abnormal while in a rendezvous. 9.10(6) says:

> If a task that calls an entry becomes abnormal while in a rendezvous, its termination does not take place before the completion of the rendezvous.

Although the calling task cannot terminate before the rendezvous is finished, 9.10(6) allows completion of the calling task while the rendezvous is being performed, i.e., although the storage associated with the calling task cannot be reclaimed while the rendezvous is in progress, the task can be marked as requiring no further execution. If an exception is raised in the rendezvous, it is sent to the calling task after it has become abnormal and after it has become complete:

```
task CALLER;

task body CALLER is
     task SERVER is
          entry REQ;
     end SERVER;
```

```
          task body SERVER is
          begin
               accept REQ do
                    abort CALLER;
                    raise PROGRAM_ERROR;      -- send exception to caller
               end REQ;
          end SERVER;
     begin
          SERVER.REQ;
               -- during the rendezvous, CALLER becomes abnormal and
               -- the rendezvous is completed with an exception
     exception
          when others =>
               -- some action
     end CALLER;
```

If CALLER completes abnormally during the rendezvous, what is the effect of propagating the exception?

A similar example can be constructed in which a task becomes abnormal while executing an allocator and activating a task:

```
     task body ACTIVATOR is
          ACC_TSK : ACC_SERVER;
     begin
          ACC_TSK := new SERVER;    -- aborts ACTIVATOR
```

Suppose that during the activation of SERVER, ACTIVATOR becomes abnormal (because SERVER aborts ACTIVATOR) and in addition, TASKING_ERROR is propagated to ACTIVATOR because SERVER is not successfully activated. The body of SERVER could be:

```
     task body SERVER is
          package P is end P;
          package body P is
          begin
               abort ACTIVATOR;
               raise PROGRAM_ERROR;
          end P;
```

When ACTIVATOR becomes abnormal, it can complete immediately since SERVER is not dependent on ACTIVATOR. SERVER's activation continues until PROGRAM_ERROR is raised, and then the unsuccessful activation of SERVER means TASKING_ERROR must be raised in ACTIVATOR. Since ACTIVATOR is complete, what is the expected effect?

!recommendation 86-08-12

An attempt to raise an exception in a completed task has no effect.

!discussion 86-08-12

The examples show that it is possible for an exception to be raised in a task that is complete. Since the task is complete, the intent is that the exception be ignored and have no effect.

Ada Commentary ai-00449-ra.wj downloaded on Thu May 12 14:50:02 EDT 1988

!standard 03.03.02 (06) 87-06-18 AI-00449/04
!class ramification 86-09-10
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (8-0-0) 86-09-10 (pending editorial review)
!status received 86-08-06
!references AI-00308, AI-00007, 83-00787
!topic Evaluating default discriminant expressions

!summary 86-09-13

Default discriminant expressions are not evaluated when a subtype indication is elaborated.

!question 86-09-13

Are default discriminant expressions evaluated (and checked for compatibility) when a subtype indication is elaborated? Consider the following examples:

```
type R (D : INTEGER := -1) is
     record
          COMP : STRING (D .. 10);
     end record;

type T1 is array (1..5) of R;    -- CONSTRAINT_ERROR? (no)

type T2 (D : BOOLEAN) is
     record
          C1 : R;                -- CONSTRAINT_ERROR? (no)
     end record;

OBJ1 : T1;                       -- CONSTRAINT_ERROR
```

Is the default discriminant expression for R evaluated when the type declarations for T1 and T2 are elaborated, or is the default expression only evaluated (and checked for compatibility) when the object declaration is elaborated?

!response 86-09-13

3.2.1(6) says:

> The elaboration of a subtype indication creates a subtype. If the subtype indication does not include a constraint, the subtype is the same as that denoted by the type mark.

3.2.1(6) goes on to say that if a constraint is present, the constraint is

elaborated and then checked for compatibility. Hence, if no discriminant constraint is present, no default expressions are evaluated when a subtype indication is elaborated. In particular, when the type declarations given in the question are elaborated, no default discriminant expressions are evaluated.

A default discriminant expression is only evaluated when an object is created. 3.2.1(6) says:

> If [an] object declaration includes an explicit initialization, the initial value is obtained by evaluating the corresponding expression. Otherwise any implicit initial values for the object or for its subcomponents are evaluated.

3.2.1(14) says:

> In the case of a component that is itself a composite object and whose value is defined neither by an explicit initialization nor by a default expression, any implicit initial values for components of the composite object are defined by the same rules as for a declared object.

Hence, when the declaration of OBJ1 in the example is elaborated, 3.2.1(6 and 14) cause the default expression for the discriminant to be evaluated (once for each array component). Each default discriminant value is then checked for compatibility (AI-00308 requires the check; AI-00007 defines how to perform the check).

In short, if a type with default discriminants is used without a discriminant constraint in a subtype indication, elaboration of the subtype indication does not cause the default expressions to be evaluated. However, if the subtype indication occurs in an object declaration that has no initialization expression, the default expressions will be evaluated as part of the object declaration's elaboration. (Similarly, occurrence of such a subtype indication in an allocator will cause the default expressions to be evaluated when the allocator is evaluated; see 4.8(6).)

Ada Commentary ai-00468-cr.wj downloaded on Thu May 12 14:50:03 EDT 1988

!standard 04.01.03 (15) 87-08-06 AI-00468/04
!class correction 86-10-10
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-10
!status received 86-10-02
!references 83-00802
!topic Correction to AI-00187/04 discussion

!summary 86-10-13

The discussion section of AI-00187/04 should be corrected to point out that a name declared by a renaming declaration is not allowed as the prefix of an expanded name if the selector is not declared in the visible part of the package denoted by the prefix.

!question 86-10-13

The discussion of this AI gives the following example:

```
package FRED is
     A : INTEGER;
     package JIM renames FRED;
end FRED;
```

The discussion points out that we can refer to FRED.A and FRED.JIM.A, or FRED.JIM.JIM.A because:

> "FRED.JIM names a package enclosing the declaration of JIM and A, and so does FRED.JIM.JIM, etc."

This reason is incorrect, since it suggests that if A were not declared in the visible part, FRED.JIM.A would also be allowed within package FRED. However, since JIM is declared by a renaming declaration, it is not in general allowed as the prefix of an expanded name [4.1.3(18)].

!recommendation 86-10-10

Correct the discussion of AI-00187/04 by replacing:

> "(since FRED.JIM names a package enclosing the declaration of JIM and A, and so does FRED.JIM.JIM, etc.)."

with

> "(since both A and JIM are declared in the visible part of a package and therefore can be denoted by an expanded name, even when the prefix is declared by a renaming declaration; see

AI-00016. FRED.JIM.A would be illegal if A were not declared in FRED's visible part; similarly FRED.JIM.JIM would be illegal if JIM were not declared in FRED's visible part)."

!discussion 86-10-10

The only reason FRED.JIM.A is legal is because of AI-00016, which allows the use of JIM as the prefix of an expanded name as long as the selector is declared in the visible part of the package denoted by the prefix.

Ada Commentary ai-00471-cr.wj downloaded on Fri May 13 09:45:02 EDT 1988

!standard 03.05.09 (09) 87-08-06 AI-00471/04
!class correction 86-10-13
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-13
!status received 86-10-13
!references 83-00822
!topic Correction to AI-00144/08 examples

!summary 86-10-13

In the examples, the value of DELTA is given incorrectly as an integer expression, 2**(-15); the expression should be 2.0**(-15).

!question 86-10-13

The expression following DELTA in the examples is 2**(-15), which has an integer type; a real type is required.

!recommendation 86-10-13

Correct the examples by replacing 2**(-15) with 2.0**(-15).

!discussion 86-10-13

The expression following DELTA in the examples must have a real type.

Ada Commentary ai-00483-cr.wj downloaded on Fri May 13 09:50:01 EDT 1988

!standard 12.03 (05) 87-08-06 AI-00483/04
!class correction 86-10-13
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-15
!status received 86-10-13
!references 83-00821
!topic Correction to question in AI-00409/03

!summary 86-10-13

The instantiation of SET_OF in the question is illegal because the formal generic type is an integer type. The formal type should be declared as a discrete type.

!question 86-10-13

The question in AI-00409/03 contains this example:

```
generic
    type T is range <>;
package SET_OF is
    type SET is array (T) of BOOLEAN;
end SET_OF;

package CHARACTER_SET is new SET_OF (CHARACTER);
subtype CHAR_SET is CHARACTER_SET.SET;
```

The instantiation of SET_OF is illegal because the formal generic parameter requires an INTEGER type.

!recommendation 87-06-04

The generic parameter declaration in the example given in the question should be changed to read:

```
type T is (<>);
```

!discussion 86-10-13

To allow an instantiation with type CHARACTER, the formal parameter must have a discrete type, not an integer type.

!standard 14.03.01 (04) 87-08-06 AI-00486/04
!class correction 86-10-13
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-15
!status received 86-10-13
!references 83-00824
!topic Correction to AI-00047/06 example

!summary 86-10-15

The call to SET_PAGE_LENGTH in the question's example should refer to file FT, not the default output file.

!question 86-10-15

The question in AI-00047/06 contains the following example:

```
CREATE (FT, OUT_FILE);
SET_PAGE_LENGTH (4);
RESET (FT, OUT_FILE);  -- (1)
```

The call to SET_PAGE_LENGTH, as written, refers to the default output file. Was it intended that the call refer to file FT?

!recommendation 86-10-15

In the question, replace the call to SET_PAGE_LENGTH with:

```
SET_PAGE_LENGTH (FT, 4);
```

!discussion 86-10-15

The call to SET_PAGE_LENGTH was intended to refer to file FT.

Ada Commentary ai-00486-cr.wj downloaded on Thu May 12 16:50:01 EDT 1988

AD-A197 564

```
with TEXT_IO; use TEXT_IO;
package P is
     procedure PR;
end P;

package body P is
     procedure PR is separate;
begin
     PUT ("Text_IO is visible here");
end P;

separate (P)
procedure PR is
begin
     PUT ("Text_IO should be visible here");
end PR;
```

Nothing in 10.1.1(4) says that the context clause that applies to P's specification also applies to subunit PR, but 10.2(6) implies visibility within PR is the same as if PR's proper body appeared where its stub appears. 10.2(6) implies the context clause for P's specification was intended to apply to all subunits of P's body.

Ada Commentary ai-00232-co.wj downloaded on Thu May 19 16:31:50 EDT 1988

!standard 07.04.01 (03) 86-07-23 AI-00232/05
!class confirmation 85-09-04
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-04
!status work-item 84-04-08
!status received 84-03-13
!references 83-00341
!topic Full declarations that implicitly declare unconstrained types

!summary 85-04-08

A full declaration of a private type can declare a constrained array subtype or a constrained type with discriminants.

!question 85-09-20

7.4.1(3) says: The type declared by the full type declaration (the full type) ... must not be an unconstrained array type." A declaration of a constrained array type implicitly declares an unconstrained TYPE:

```
package P is
     type T is private;
private
     type T is array (1..5) of INTEGER;    -- unconstrained type
                                           -- declared implicitly
end P;
```

Wasn't the intent probably to say that the SUBTYPE declared by the full type declaration must not be an unconstrained array type?

A similar situation occurs for types with discriminants, as in:

```
pack P is
     type T is private;
private
     type R (D : INTEGER) is
          record ... end record;
     type T is new R (3);                  -- unconstrained type
                                           -- declared implicitly
end P;
```

Here the derived base type is unconstrained, but the subtype T is constrained. Is this full declaration of T legal?

!response 85-09-20

3.3.1(4) says:

> The elaboration of the type definition for a numeric or derived type creates both a base type and a subtype of the base type; the same holds for a constrained array definition.

Thus, a full declaration for a private type sometimes creates a base type and a different subtype. In such a case, 3.3.1(5) says the declared identifier denotes the subtype. The current wording of 7.4.1(3) talks about "the type" declared by the full type declaration, seemingly leaving unclear whether the base type or the subtype is meant. But 7.4.1(2) says:

> A private type declaration and the corresponding full type declaration define a single type.

i.e., the type declared by a full type declaration is the type denoted by the name explicitly declared by the declaration. When the full type declaration declares a constrained array, the declared name denotes the array subtype [3.3(5)], and similarly, when the full type declaration derives from a constrained type with discriminants, the declared name denotes the (constrained) subtype. In neither case does the declared name denote an unconstrained array type or an unconstrained type with discriminants, so both declarations given in the example are legal.

A full type declaration such as

```
type T is array (POSITIVE range <>) of INTEGER;
```

or

```
type T is new R;
```

would be illegal because the entity denoted by T (i.e., the full type) would be an unconstrained array type or an unconstrained type with discriminants.

Ada Commentary ai-00234-bi.wj downloaded on Thu May 19 16:31:51 EDT 1988

!standard 03.05.05 (10) 87-03-16 AI-00234/05
!class binding interpretation 84-04-13
!status approved by WG9/AJPO 87-03-10
!status approved by Director, AJPO 87-03-10
!status approved by Ada Board (22-0-0) 87-02-19
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-04
!status work-item 85-02-04
!status received 84-04-13
!references 83-00347
!topic Lower bound for 'IMAGE of enumeration values

!summary 87-02-23

The lower bound of the string returned by the predefined attribute IMAGE is one.

!question 87-02-23

What is the lower bound of the string returned by 'IMAGE for an enumeration type?

!recommendation 87-02-23

The lower bound of the string returned by the predefined attribute IMAGE is one.

!discussion 85-02-04

3.5.5(10) specifically says that when the prefix of 'IMAGE has an integer type, the lower bound of the image is one. This statement was intended to apply to the result for enumeration types as well.

Ada Commentary ai-00236-bi.wj downloaded on Tue May 10 18:45:02 EDT 1988

!standard 10.05 (04) 86-07-23 AI-00236/12
!class binding interpretation 85-05-17
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9-1-2) 85-11-20 (by ballot)
!status committee-approved (8-1-0) 85-09-04 (pending letter ballot)
!status failed letter ballot (5-6-1) 85-09-04
!status committee-approved (9-0-2) 85-05-17
!status work-item 85-02-04
!status received 84-04-13
!references 83-00350, 83-00591, 83-00617, 83-00660, 83-00672
!topic Pragma ELABORATE for bodiless packages with tasks

!summary 85-07-25

If a package declares a task object but no package body is required or provided by a programmer, 9.3(5) says an implicit body is provided. The effect of a pragma ELABORATE that names such a package is to require that the implicitly provided package body be elaborated, thereby activating the task declared in the package.

!question 85-09-14

Consider this example:

```
with TASK_TYPES;
pragma ELABORATE (TASK_TYPES);   -- to prevent PROGRAM_ERROR
package P is
     TSK : TASK_TYPES.SOME_TASK_TYPE;
end P;

with P;
pragma ELABORATE (P);    -- requires elaboration of P's implicit body
package Q is ...
```

Assume that P does not later have a body supplied. 10.5(4) says P "must have a library unit body". Since P has no body, this would seem to imply that the pragma ELABORATE should be ignored [2.8(9)]. However, 9.3(5) says there is an implicit package body for P that causes activation of the task declared within P. Thus the above example might be intended to insure that the task TSK is activated before Q's elaboration. Is the implicit body elaborated?

!recommendation 85-07-25

If a pragma ELABORATE names a package that has an implicitly provided body, the implicit body is elaborated before the compilation unit associated with the pragma.

!discussion 85-09-14

9.3(5) says, regarding the rules for activating tasks:

> For the above rules, ... for any package without a package body, an implicit package body containing a single null statement is assumed.

10.5(4) says, with respect to the pragma ELABORATE:

> Each argument of such a pragma must be the simple name of a library unit mentioned by the context clause, and this library unit must have a library unit body.

The basic question is: can the pragma ELABORATE be obeyed for a package that declares a task object but that has no explicitly provided package body?

It considering this question, it should be noted that 10.5(4) uses the term "library unit body", which, by 1.5(6) is a reference to the syntactic term, library_unit_body. This reference to a syntactic term does not imply the body must be provided explicitly by a programmer, since the rules of the Standard apply equally well to implicitly and explicitly declared units. For example, 10.5(1-2) specifies that before execution of the main program all library unit bodies must be elaborated in a certain order. These rules also use the term "library unit body," and here, the intent is clearly to cover the bodies implicitly declared by 9.3(5) as well as those explicitly declared by a programmer.

Also, although 9.3(5) says "For the above rules" and thereby seems to limit the effect of the paragraph in some way that might be understood to exclude rules given in 10.5, such an interpretation is untenable, since the elaboration order rules given in 10.5(1-2) must be considered to apply to implicit package bodies created by 9.3(5).

In short, it would be unreasonable to interpret 10.5(1-2) as applying to implicit as well as explcit package bodies, and then to interpret identical phrases in 10.5(4) to apply only to explicitly declared package bodies.

In addition, it is important that the pragma be allowed to apply to implicitly generated bodies since a programmer cannot always know when a task object is being created. When an object having a limited private type is declared in a package, a task can be declared without the programmer's knowledge. It would be a nuisance (to say the least) to have to provide an explicit dummy package body just to ensure that any potential task will be activated by a pragma ELABORATE.

Finally, it is always harmless to name a package that has no (explicit or implicit) body, since the pragma will then be ignored [2.8(9)]; at most, an implementation will produce a warning message.

In short, when the pragma ELABORATE names a package that has an implicitly provided body, it is both useful and consistent with the Standard to require

that the implicit body be elaborated before the compilation unit containing the pragma.

Ada Commentary ai-00237-bi.wj downloaded on Tue May 10 18:45:03 EDT 1988

!standard 12.03 (17) 86-07-23 AI-00237/06
!standard 09.03 (05)
!class binding interpretation 84-04-13
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (8-0-0) 86-02-21
!status work-item 85-02-04
!status received 84-04-13
!references 83-00348, 83-00591, 83-00708
!topic Instances having implicit package bodies

!summary 86-04-02

Given a generic package declaration that does not require a body and that has no explicit body, when the generic package is instantiated, if the instance specification requires a body, then an implicit instance body is created and is elaborated when the instantiation is elaborated.

!question 86-03-11

Consider the following sequence of compilations:

```
with TASK_TYPES;
generic
package SEMAPHORE_TASKS is
     S1, S2 : TASK_TYPES.SEMAPHORE_TYPE;  -- declare two task objects
end SEMAPHORE_TASKS;

with SEMAPHORE_TASKS;
procedure MAIN is
     I : INTEGER;
     package SEM is new SEMAPHORE_TASKS;
     -- (1)
     J : INTEGER;
     -- (2)
begin
     null;
end MAIN;
```

Note that SEMAPHORE_TASKS is not required to have a body. If the user doesn't supply a body for SEMAPHORE_TASKS, then the compiler must create an implicit one for SEM, in order to activate the tasks SEM.S1 and SEM.S2. Where is this implicit body elaborated, i.e., are SEM.S1 and SEM.S2 activated at point (1) (as implied by 12.3(17)) or at point (2) (as implied by 9.3(5))?

The same problem also occurs when task types are passed as actual generic parameters to a generic package that otherwise has no tasks, as in:

```
generic
     type LP is limited private;
package P is
     OBJ : LP;
end P;

with P, TASK_TYPES;
package Q is
     package NP is new P (TASK_TYPES.SEMAPHORE_TYPE);
     -- (1)
     K : INTEGER;
     -- (2)
end Q;
```

Again, an implicit body is needed to activate the task NP.OBJ, but is this body elaborated at point (1) or at point(2)?

!recommendation 86-04-02

Given a generic package declaration that does not require a body and that has no explicit body, when the generic package is instantiated, if the instance specification requires a body, then an implicit instance body is created and is elaborated when the instantiation is elaborated.

!discussion 86-03-11

9.3(5) says:

> For any package without a package body, an implicit package body containing a single null statement is assumed If a package without a package body is declared immediately within some program unit or block statement, the implicit package body occurs at the end of the declarative part...

12.3(17) says:

> For the elaboration of a generic instantiation...the implicitly generated instance is elaborated.

From 9.3(5), it is clearly intended that package instances should have implicit bodies so the task objects can be activated. However, the phrase "the implicitly generated instance" in 12.3(17) is intended to refer to both the instance specification and the instance body, i.e., any body associated with an instance is to be elaborated when the instantiation is elaborated. In particular, the intent is that any implicit instance bodies be elaborated at the points of instantiation (points (1)), and hence, task objects declared within the instances are to be activated at points (1).

Ada Commentary ai-00239-nb.wj downloaded on Tue May 10 18:50:02 EDT 1988

!standard 14.03.09 (06)
!standard 14.03.09 (09)
!standard 03.05.05 (11)
!class non-binding interpretation 85-09-04
!status approved by WG9/AJPO 87-02-20
!status approved by Director, AJPO 87-02-20
!status approved by Ada Board (?1-0-0) 87-02-19
!status approved by WG9 86-05-09
!status committee-approved (10-0-0) 85-09-04
!status failed letter ballot (7-4-1)
!status committee-approved (9-2-0) 85-05-18 (pending letter ballot)
!status work-item 85-03-04
!status returned to committee by WG9/Ada Board 85-02-26
!status committee-approved 84-11-26
!status work-item 84-06-12
!status received 84-04-13
!references 83-00345, 83-00477, 83-00623, 83-00622
!topic ENUMERATION_IO and IMAGE for non-graphic characters

!summary 86-05-20

If ENUM_IO is an instantiation of ENUMERATION_IO for a character type that contains a non-graphic character, e.g.,

```
package ENUM_IO is new ENUMERATION_IO (CHARACTER);
```

then for each non-graphic character (such as ASCII.NUL), ENUM_IO.PUT should output the corresponding sequence of characters used in the type definition (e.g., PUT(ASCII.NUL) should output the string "NUL" if SET has the value UPPER_CASE and WIDTH is less than 4). Furthermore, ENUM_IO.GET should be able to read the sequence of characters output by ENUM_IO.PUT for a non-graphic character, returning in its ITEM parameter the corresponding enumeration value.

Similarly, the image of a non-graphic character (i.e., the result returned for the attribute designator IMAGE) should be the sequence of characters used in the type definition of CHARACTER (e.g., CHARACTER'IMAGE(ASCII.NUL) = "NUL"), and 'VALUE should accept such a string as representing the corresponding enumeration value.

An implementation conforms to the Standard in this respect if the result produced by 'IMAGE for a non-graphic character is accepted by 'VALUE, and if the result (if any) produced by PUT can be read by GET; GET is also allowed to raise DATA_ERROR when attempting to read any string produced by PUT for a non-graphic character.

This interpretation is non-binding, i.e., implementers are encouraged to conform to it but are not required to do so by the validation tests. A future version of the Standard may incorporate this interpretation.

!question 85-01-03

When ENUMERATION_IO is instantiated for the predefined CHARACTER type, what output is allowed for non-graphic characters such as ASCII.NUL? Can GET read a string representing a non-graphic character without raising DATA_ERROR?

!recommendation 86-01-20

The result of executing the PUT operation of ENUMERATION_IO for a non-graphic character value should be the same as if the corresponding value in type CHARACTER had been declared as an identifier.

For non-graphic characters, the GET operation for ENUMERATION_IO should be able to read the sequence of characters output by the PUT operation; GET then returns in its ITEM parameter the corresponding enumeration value.

The image of a non-graphic character should be an upper case sequence of characters corresponding to the sequence used in the declaration of type CHARACTER.

!discussion 86-05-20

Non-graphic characters (the ASCII "control characters") are declared in Appendix C with italicized letters. C(12) explains the use of italics:

> [The] literals corresponding to control characters are not identifiers; they are indicated in italics in this definition.

Since an enumeration literal must be either an identifier or a character literal [3.5.1(2)], and since non-graphic characters are not declared with identifiers or with character literals, no enumeration literal, and in particular, no identifier declared in the package STANDARD has the value of a non-graphic character. (For example, the non-graphic character, ASCII.NUL cannot be named as STANDARD.NUL.)

Because no enumeration literal has the value of a non-graphic character, difficulties arise when attempting to satisfy what the Standard requires for the IMAGE and VALUE attributes. The definition for T'IMAGE says [3.5.5(11)]:

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

However, the definition for T'VALUE says [3.5.5(13)]:

> For an enumeration type, if the sequence of characters has the syntax of an enumeration literal and if this literal exists for the base type of T, the result is the corresponding enumeration value. ... In any other case, the exception CONSTRAINT_ERROR is raised.

If CHARACTER'IMAGE(ASCII.NUL) produces the string "NUL" then CHARACTER'VALUE

("NUL") must apparently raise CONSTRAINT_ERROR because no enumeration literal NUL exists for type CHARACTER; no such identifier was declared for type CHARACTER. If CHARACTER'IMAGE(ASCII.NUL) outputs just the character code for ASCII.NUL, CHARACTER'VALUE must still raise CONSTRAINT_ERROR:

```
MY_NUL : STRING(1..1) := (1 => ASCII.NUL);
MY_CHR : CHARACTER    := CHARACTER'VALUE(MY_NUL); -- CONSTRAINT_ERROR
```

Even if CHARACTER'IMAGE(ASCII.NUL) = MY_NUL, CHARACTER'VALUE(MY_NUL) must raise CONSTRAINT_ERROR because the value of MY_NUL does not satisfy the syntax for an enumeration literal. In short, there is no sequence of characters that can be output by IMAGE for a non-graphic character that will satisfy both the syntax of an enumeration literal and be a literal that exists for the base type. So T'VALUE seemingly must raise CONSTRAINT_ERROR for whatever T'IMAGE produces for a non-graphic character despite 3.5.5(11)'s requirement that IMAGE and VALUE be inverse functions.

A similar problem exists for PUT and GET in ENUMERATION_IO. The definition of PUT in 14.3.9(9) says:

> Outputs the value of the parameter ITEM as an enumeration literal (either an identifier or a character literal) ...

As we have already noted, there is no identifier or character literal that corresponds to a non-graphic character. The Standard could be interpreted here to mean that the output for a non-graphic character is implementation-dependent. But can GET read what PUT produces? The definition of GET in 14.3.9(6-7) says:

> ... reads an identifier according to the syntax of this lexical element ... or a character literal according to the syntax of this lexical element. ...
>
> The exception DATA_ERROR is raised if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype ENUM.

Clearly, the same problem arises for GET as for the VALUE attribute; no string produced by PUT can satisfy the requirements imposed on GET, so it appears that GET must raise DATA_ERROR whenever it attempts to read what PUT produces for a non-graphic character. (Alternatively, PUT might produce no output for non-graphic characters.)

Despite these problems of interpretation, the intent of the Standard seems clear: CHARACTER'IMAGE for a non-graphic character should produce a string that CHARACTER'VALUE will accept without raising CONSTRAINT_ERROR. Similarly, GET should be able to read the string produced by PUT for a non-graphic character. Given this intent, how should the Standard be interpreted? One alternative is to allow IMAGE and PUT to output implementation-defined character sequences for non-graphic characters.

If implementation-defined character sequences are output, should the

sequences satisfy the syntax for an enumeration literal? If not, then sequences such as "'NUL'", "\000", and "'<NUL>'" might be chosen by different implementations. In favor of this interpretation is the statement in 3.5.5(11) that the image of a non-graphic character "is implementation-defined." This approach would amount to extending the syntax of enumeration literals for purposes of interpreting the behavior of the attribute VALUE and the operation GET.

Extending the syntax, however, would raise other questions, e.g., if an implementation outputs 'NUL' for ASCII.NUL, where should reading stop when GET is reading a string such as 'NU'? Normally reading would stop after 'N since the U does not satisfy the syntax of a character literal, but if the syntax has been extended to include a special form for non-graphic characters, should reading stop after the second apostrophe? Moreover, it is not necessary to extend the syntax to have a representation for non-graphic characters. The syntax of identifiers serves quite adequately, e.g., ASCII.NUL could be represented as NUL.

Identifier syntax can be used to represent non-graphic character values because non-graphic characters are only defined in the predefined type CHARACTER and in types derived, directly or indirectly, from this type. Since the predefined CHARACTER type contains no identifiers as enumeration literals, any identifier associated with a non-graphic character can never conflict with a predefined enumeration literal of type CHARACTER.

In short, the solution that both satisfies the intent of the language and is least likely to cause additional problems is to require that IMAGE and PUT produce strings satisfying identifier syntax, and that VALUE and GET accept these strings as correct representations of the corresponding non-graphic characters.

Given that identifier syntax is to be used, is the choice of identifiers implementation-dependent? Although all members of the Language Maintenance Committee think it is reasonable to interpret the Standard as requiring identifier syntax, some feel the Standard cannot be interpreted to require any particular association of identifiers with non-graphic characters. Others believe that even though identifiers are not, technically speaking, used to declare the non-graphic characters in CHARACTER's declaration, once it is understood that IMAGE and PUT output identifiers to represent non-graphic characters, it is reasonable to say that the character sequences given in CHARACTER's declaration determine what identifiers are output (and correspondingly, what identifiers are accepted by VALUE and GET). There is no advantage to be gained by leaving the choice of identifiers up to an implementation. Moreover, if the choice is left implementation dependent, then the visible representation of non-graphic characters will be implementation-dependent for Ada's language defined character type, but not for any other character type defined by a user (e.g., since a character type called EBCDIC must be defined using identifiers and character literals, the visible representation of EBCDIC's "non-graphic" characters will be completely defined). For these reasons, it seems reasonable to specify that the strings output by IMAGE and PUT for non-graphic characters be considered completely defined by the Standard.

In short, the output produced by IMAGE and PUT for a non-graphic character should correspond to the character sequence given in the declaration of type CHARACTER, and VALUE and GET should accept this sequence of characters as a string representing the corresponding non-graphic character.

Ada Commentary ai-00240-ra.wj downloaded on Tue May 10 18:50:03 EDT 1988

!standard 03.05.04 (03) 86-07-23 AI-00240/05
!class ramification 85-09-04
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (6-0-3) 85-09-04
!status work-item 84-06-12
!status received 84-05-03
!references 83-00357
!topic Integer type definitions cannot contain a RANGE attribute

!summary 85-09-20

A range attribute is not allowed in an integer type definition.

!question 85-09-20

The syntax rules permit the following declaration:

```
type INT is range T'RANGE;
```

where T is declared as an array type. 3.5.4(3) requires that "each bound of the range [in an integer type definition] must be defined by a static expression of some integer type." Does this imply that T'RANGE is legal if and only if the bounds for T are declared with static integer expressions, or was it the intent to forbid the use of T'RANGE in this context?

!response 85-09-20

3.5.4(3) says "If a range constraint is used as an integer type definition, each bound of the range must be defined by a static expression of some integer type ..." The range attribute T'RANGE is defined to "yield the range T'FIRST .. T'LAST" [3.6.2(7)] and T'FIRST is not a static expression, so a range attribute is not allowed in an integer type definition.

Ada Commentary ai-00242-bi.wj downloaded on Wed May 11 08:45:02 EDT 1988

!standard 06.03.02 (03) 87-06-18 AI-00242/09
!standard 02.08 (09)
!class binding interpretation 86-03-26
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (18-1-4) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (8-0-0) 86-09-10 (pending editorial review)
!status work-item 86-09-10
!status failed letter ballot (6-4-3) 86-09
!status committee-approved (8-1-1) 86-05-13 (pending letter ballot)
!status work-item 86-03-26
!status received 84-05-03
!references 83-00351, 83-00637, 83-00726, 83-00797
!topic Subprogram names allowed in pragma INLINE

!summary 86-09-17

If the pragma INLINE appears at the place of a declarative item, every name in the pragma must denote at least one subprogram or generic subprogram declared explicitly earlier in the same declarative part or package specification. If the pragma appears after a given library unit, the pragma must contain just the name of the library unit, and the library unit must be a subprogram or a generic subprogram.

If a pragma INLINE appears at the place of a declarative item and a name in the pragma is overloaded, the pragma applies just to those subprograms whose declarations occur (explicitly) earlier in the same declarative part or package specification.

If a name in a pragma INLINE is declared by a renaming declaration, and the denoted subprogram is explicitly declared earlier in the same declarative part or package specification, inline expansion is desired for every call of the denoted subprogram (whether the call uses the new or the old name).

If a pragma INLINE applies to a subprogram, inline expansion is desired for every call of the subprogram, whether or not the call uses a name declared by a renaming declaration.

!question 86-08-05

There are several questions regarding the interpretation of names appearing as arguments of pragma INLINE.

First, 2.8(9) says:

> a pragma ... has no effect if ... its arguments do not correspond to what is allowed for a pragma.

6.3.2(3) says that for a pragma INLINE appearing at the place of a declarative item:

> each name [given as an argument] must denote a subprogram or generic subprogram declared by an earlier declarative item of the same declarative part or package specification. ... If the pragma appears after a given library unit, the only name allowed is the name of this unit.

Suppose not all the names given in the pragma satisfy these requirements, e.g.,

```
declare
     procedure P;
     X : INTEGER;
     pragma INLINE (P, X);
```

Since X is not even a subprogram, is the pragma to have no effect for P? Also consider this example:

```
procedure Y;                -- a library unit
procedure Z;                -- a library unit
pragma INLINE (X, Y, Z);
```

Since there is no visible X, and since the name of the preceding library unit is not Y, is the pragma considered to apply just to Z, or does it have no effect at all?

Similar questions arise when an argument of pragma INLINE is overloaded and some of the denoted subprograms are not declared earlier in the same declarative part or package specification, e.g.,

```
procedure P (B : BOOLEAN) is            -- P.1
begin
     procedure P (C : CHARACTER) is     -- P.2
     begin ... end P;
     pragma INLINE (P);
```

The name in the pragma can denote either P.1 or P.2, but P.1 is not declared earlier in the same declarative part as the pragma. Is the pragma to be ignored, since one of the denotable subprograms does not satisfy the pragma's requirements, or is the pragma to be applied just to those denotable subprograms that satisfy the requirements, i.e., is the effect of the pragma to request inline expansion only for calls to P.2?

Next, 6.3.2(3) speaks of names that "denote" a subprogram. Consider the effect of this term on names declared by a renaming declaration:

```
procedure M;
package P is
     procedure NEW_M renames M;
     pragma INLINE (NEW_M);
```

Since the name NEW_M denotes subprogram M, which is not declared in package P, does the pragma fail to apply to subprogram calls that use the name NEW_M?

Suppose the situation is reversed, i.e., the pragma applies to procedure M but not to NEW_M. Are calls using the name NEW_M to be expanded inline because the IN_LINE pragma applies to the denoted subprogram, or does the pragma for subprogram M have no effect on calls using the name NEW_M?

Finally, there are different kinds of subprogram. In particular, enumeration literals and function attributes (e.g., T'SUCC) are subprograms. What does it mean if such subprogram names appear in a pragma INLINE? Also, can the pragma be applied to implicitly declared subprograms such as derived subprograms and predefined operators?

To summarize the questions:

1. When the pragma contains several names, not all of which satisfy the requirements, does the pragma apply just to those names that do satisfy the requirements, or is it completely ignored?

2. What is the effect if a name is declared as an enumeration literal, function attribute, derived subprogram, or predefined operator?

3. When a name is overloaded, does the pragma apply just to those subprograms declared earlier in the same declarative part or package specification?

4. For names declared by a renaming declaration,

   Does the applicability of the pragma depend on where the renaming declaration appears or where the declaration of the denoted subprogram appears?

   Is the effect of the pragma to request inline expansion just for calls that use the name declared by the renaming declaration?

   If a pragma INLINE applies to the denoted subprogram but not to the name declared by a renaming declaration, is the effect to request that calls using the new name not be expanded inline?

!recommendation 86-09-17

If the pragma INLINE appears at the place of a declarative item, every name in the pragma must denote at least one subprogram or generic subprogram declared explicitly earlier in the same declarative part or package specification. If the pragma appears after a given library unit, the pragma must contain just the name of the library unit, and the library unit must be a subprogram or a generic subprogram.

If a pragma INLINE appears at the place of a declarative item and a name in the pragma is overloaded, the pragma applies just to those subprograms whose declarations occur (explicitly) earlier in the same declarative part or package specification.

If a name in a pragma INLINE is declared by a renaming declaration, the

denoted subprogram must be explicitly declared earlier in the same declarative part or package specification. Inline expansion is desired for every call of the denoted subprogram (whether the call uses the new or the old name).

If a pragma INLINE applies to a subprogram, inline expansion is desired for every call of the subprogram, whether or not the call uses a name declared by a renaming declaration.

!discussion 86-07-07

When more than one name is given in a pragma INLINE, the intent is that every name denote at least one subprogram or generic subprogram declared explicitly earlier in the same declarative part or package specification. Therefore, in the example where one of the names denotes an object instead of a subprogram, the pragma has no effect (2.8(9)). Similarly, if there are no visible declarations corresponding to a name that appears in the pragma, the pragma has no effect.

If a name given in the pragma is overloaded but only some of the visible declarations occur earlier in the same declarative part or package specification, the intent is to allow the pragma to be obeyed for those names whose declarations do occur earlier in the allowed places.

6.3.2(3) says:

> each name must denote a subprogram or a generic subprogram ...

This wording suggests that applicability of the pragma is determined by the nature of the denoted entity. If a name declared by a renaming declaration is used in the pragma, the denoted entity cannot be a generic subprogram (since a generic subprogram cannot be renamed). If the renamed entity is a subprogram, applicability of the pragma is determined by the nature of the denoted subprogram, i.e., the denoted subprogram must have been declared explicitly by a subprogram declaration that occurs earlier in the same declarative part or package specification. If the renaming declaration declares an overloaded name, then at least one of the denoted subprograms must have been declared appropriately.

Since the pragma INLINE applies to denoted subprograms, inline expansion is desired for every call of the denoted subprogram (whether a call uses the new or the old name, and regardless of which name was used in the pragma).

Since an attribute is an implicitly declared operation, a pragma INLINE that contains (or denotes only) an attribute has no effect. For example:

```
type INT is range 1..10;
function SUCC (X : INT) return INT renames INT'SUCC;
pragma INLINE (INT'SUCC);        -- no effect
pragma INLINE (SUCC);            -- no effect
```

The pragmas have no effect even though INT'SUCC has the correct syntactic form for a name and even though the denoted entity is a subprogram.

If a name in a pragma INLINE denotes an attribute, an enumeration literal, a predefined operator, or a derived subprogram, the pragma must also denote a subprogram that is declared explicitly earlier in the same declarative part or package specification; otherwise the pragma has no effect.

Ada Commentary ai-00243-co.wj downloaded on Wed May 11 08:45:03 EDT 1988

!standard 14.03.05 (07) 86-12-01 AI-00243/05
!class confirmation 86-04-15
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (8-0-1) 86-05-12 (pending editorial review)
!status work-item 86-03-26
!status received 84-05-03
!references 83-00353, 83-00353
!topic Overriding width format in TEXT_IO

!summary 86-04-16

If the specification of WIDTH or FORE in a call of PUT is insufficiently large, the output is given with no leading spaces.

!question 86-04-16

14.3.5(7) states what happens if there is not enough space for a value to be output in the format specified by a width parameter:

> The format given for a PUT procedure is overridden if it is insufficiently wide.

Unfortunately, this paragraph does not say how the format is overridden. Is the insufficient width replaced by the minimum width needed to accomodate the actual VALUE being output in this call, or is it replaced with the DEFAULT width, which is sufficient for any value of the type?

!response 86-09-05

For the output of integer values, 14.3.7(10) states:

> Outputs the value of the parameter ITEM as an integer literal, with no underlines, no exponent, and no leading zeros (...), and a preceding minus sign for a negative value.

The WIDTH parameter is only used to decide if additional leading spaces should be output (14.3.7(11)):

> If the resulting sequence of characters to be output has fewer than WIDTH characters, then leading spaces are first output to make up the difference.

Similarly, the FORE parameter for outputting real values and the WIDTH parameter for outputting enumeration values only determine whether leading or trailing spaces (respectively) should be output (see 14.3.8(13-15) and 14.3.9(9)). So if the specified output format has insufficient width, the output contains no leading spaces, but is otherwise unaffected.

Ada Commentary ai-00244-bi.wj downloaded on Wed May 11 08:50:02 EDT 1988

!standard 04.03.01 (01) 86-12-01 AI-00244/04
!class binding interpretation 86-01-28
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (9-0-0) 86-05-13 (pending editorial review)
!status work-item 85-04-09
!status received 84-05-14
!references 83-00360
!topic Record aggregates with multiple choices in a component association

!summary 86-07-03

In a record aggregate, a component association having multiple choices denoting components of the same type is considered equivalent to a sequence of single choice component associations representing the same components.

!question 86-07-07

Section 4.3.1(1) makes clear that when more than one record component is specified in a single component association, the components must all be of the same type. Section 4.3.1(3) goes on to require that each expression is evaluated once for each associated component and its value is checked against the subtype of the respective component.

Now consider:

```
N : INTEGER := ...;

type REC is
     record
          S1 : STRING (1 .. 9);
          S2 : STRING (1 .. N);
     end record;

OBJ1 : REC := (S1 => (1 => '1', others => '*'),        -- legal
               S2 => (1 => '1', others => '*'));       -- illegal by
                                                       --  4.3.2(3)

OBJ2 : REC := (S1 | S2 => (1 => '1', others => '*')); -- illegal?
--                        ^                      ^
--                        +--  array aggregate  --+
--                             of type STRING
```

Because component S1 has a static index constraint, the array aggregate is legal if used only as the expression for S1, as in OBJ1. Because component S2 does not have a static index constraint, the array aggregate is illegal according to 4.3.2(3) if used only as the expression for S2, as in OBJ1. Now for OBJ2, the same array aggregate is used for both components S1 and S2 in a single component association. Presumably the array aggregate is still legal for component S1 and still illegal for S2, so the initialization for OBJ2 is illegal. Is this analysis correct?

In the above example, also consider the following declaration:

```
OBJ3 : REC := (S1 | S2 => (others => '*'));
```

If N is not equal to 9, what is the applicable index constraint?

!recommendation 86-07-03

In a record aggregate, a component association having multiple choices denoting components of the same type is considered equivalent to a sequence of single choice component associations representing the same components.

!discussion 86-07-07

There are several places in the Standard where a condensed notation is considered to be equivalent to an expanded notation, and the rules are then given in terms of the expanded notation, e.g., declarations and specifications (3.2(10,11,12)). Similarly, the rules for component associations in aggregates are ultimately expressed in terms of single choices and/or single components, e.g., in 4.3.1(3) and 4.3.2(2,10,11). For both the condensed and expanded notation, the expressions in each component association are evaluated in some order and the expression for each associated component is checked for compatibility with the subtype of the subcomponent [4.3.1(3)]. In all of these places, if the expanded notation contains an illegality, then so does the condensed notation.

Consequently, the initialization for OBJ2 is intended to be equivalent to that for OBJ1; both are illegal by 4.3.2(3) because of the nonstatic constraint for component S2. The initialization for OBJ3 is legal since it is equivalent to:

```
OBJ4 : REC := (S1 => (others => '*'),    -- legal by 4.3.2(3)
               S2 => (others => '*'));   -- legal by 4.3.2(3)
```

and the applicable index constraints for the others choice are 1..9 and 1..N, respectively.

Ada Commentary ai-00247-co.wj downloaded on Wed May 11 08:50:03 EDT 1988

!standard 14.02.01 (03) 86-07-23 AI-00247/05
!class confirmation 85-09-04
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-04
!status work-item 84-06-12
!status received 84-05-14
!references 83-00359
!topic A non-null FORM argument can be required by an implementation

!summary 85-09-17

An implementation can require that a non-null FORM argument be given to CREATE and/or OPEN by raising USE_ERROR if one is not provided.

!question 85-09-17

Is an implementation allowed to specify that a non-null FORM string always be supplied to CREATE or OPEN under certain conditions, e.g., when creating a direct file for an unconstrained array or record type (in this case, the FORM string might specify the maximum element size)?

!response 85-09-17

The Standard only says for FORM that "a null string for FORM specifies the use of the default options of the implementation for the external file." The wording implies that FORM is to be used to provide additional information needed to control the creation or opening of a file, e.g., whether the file exists on tape or disk, and if on tape, the format of the file. Although the wording and parameter specification may seem to suggest that it should always be possible to call OPEN or CREATE with a null FORM parameter, no such rule is actually stated. In particular, there may be no appropriate default options for certain kinds of external files. Consequently, an implementation is free to impose additional restrictions on calls to OPEN or CREATE such that some or all calls to OPEN or CREATE will raise USE_ERROR if the FORM parameter is null. (Note: since the reasons for raising an exception depend on the characteristics of the external file being opened or created, USE_ERROR is the appropriate exception to raise; see 14.4(5).)

Ada Commentary ai-00251-bi.wj downloaded on Wed May 11 09:45:04 EDT 1988

!standard 04.09 (11) 87-03-16 AI-00251/05
!class binding interpretation 84-05-26
!status approved by WG9/AJPO 87-03-10
!status approved by Director, AJPO 87-03-10
!status approved by Ada Board (23-0-0) 87-02-19
!status approved by WG9 85-11-18
!status committee-approved (10-0-0) 85-09-04
!status work-item 84-06-12
!status received 84-05-26
!references AI-00190, 83-00371
!topic Are types derived from generic formal types static subtypes?

!summary 85-09-17

Types derived from generic formal types are not static subtypes.

!question 85-09-17

4.9(11) states, "A static subtype is ... a scalar base type, other than a generic formal type ...". 12.1(3) defines generic formal type: "The [term] ... generic formal type ... [is] used to refer to corresponding generic formal parameters." This implies that a type derived from a generic formal type is not a generic formal type. Therefore, a scalar base type which is derived from a generic formal type is a static subtype, although this does not seem to be the intent. Are types derived from generic formal types static subtypes?

!recommendation 85-09-17

A type derived (directly or indirectly) from a generic formal type is not a static subtype.

!discussion 87-02-23

A consequence of the current wording is:

```
generic
     type T is range <>;
package P is
     type S is new T;
     type F is digits S'LAST; -- not intended to be legal
end P;
```

It was intended that the bounds of static subtypes be known when the generic is compiled, i.e., a type derived (directly or indirectly) from a generic formal type should not be considered static.

Ada Commentary ai-00257-ra.wj downloaded on Wed May 11 09:45:06 EDT 1988

!standard 10.03 (09) 86-07-23 AI-00257/04
!class ramification 84-11-28
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-06-12
!status received 84-05-26
!references 83-00367
!topic Restricting generic unit bodies to compilations

!summary 85-01-04

10.3(9) allows an implementation to require that bodies and subunits of a generic unit appear in the same compilation. An implementation is allowed to apply this rule selectively, i.e., the conditions under which an implementation requires placement in a single compilation may depend on characteristics of the generic specification, body, or subunit.

!question 85-01-04

Is an implementation allowed to require that generic unit bodies (and subunits) be present in the same compilation under some but not all circumstances, i.e., can the rule in 10.3(9) be applied selectively?

!response 85-01-04

10.3(9) says:

> "An implementation may require that a generic declaration and the corresponding proper body be part of the same compilation, whether the generic unit is itself separately compiled or is local to another compilation unit. An implementation may also require that subunits of a generic unit be part of the same compilation."

The "may" in this paragraph means an implementation has a choice of whether to apply the rule or not. The rule is stated in terms of a single generic declaration. Thus an implementation is allowed to apply the rule to some generic declarations and not to others, i.e., an implementation can decide to require the presence of a body (or subunits) in some circumstances and not in others.

For example, an implementation might decide to impose the restriction just for those generic units that have a formal parameter of a private or limited private type (since instantiations of such units might be illegal, depending on how the type is used within the generic unit's body). Or an implementation might decide to impose the restriction for any generic unit that is instantiated prior to the occurrence of its proper body.

Ada Commentary ai-00258-bi.wj downloaded on Wed May 11 09:50:03 EDT 1988

!standard 13.07.02 (07) 86-11-19 AI-00258/05
!standard A (34)
!class binding interpretation 84-05-26
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status WG14/ADA Board approved 84-11-27
!status WG14-approved 84-11-27
!status board-approved 84-11-26
!status committee-approved 84-06-28
!status work-item 84-06-12
!status received 84-05-26
!references 83-00368
!topic 'POSITION etc. for renamed components

!summary 84-09-10

The prefix for 'POSITION, 'FIRST_BIT, and 'LAST_BIT must have the form R.C, where R is a name denoting a record and C is the name of a component of the record.

!question 84-09-10

Can a name declared by a renaming declaration be used with the 'POSITION, 'FIRST_BIT, and 'LAST_BIT attributes? The definitions in 13.7.2(7-10) all use the notation R.C, suggesting that the prefix of these attributes must have the form of a selected component whose prefix denotes a record, but Annex A(34) says that P'POSITION is allowed "for a prefix P that denotes a component of a record object". The wording in the annex allows a name declared by a renaming declaration as a prefix for 'POSITION, while the wording in 13.7.2 seems to disallow such usage. Which wording is correct?

!recommendation 84-09-10

The wording in the Annex is a summary of the actual definition, which is given in 13.7.2(7).

!discussion 84-09-10

The preferred interpretation from an implementer's viewpoint is to require that the prefix have the form of a selected component whose prefix denotes a record, because of examples like the following:

```
type R is record
     C1 : String (1..M);      -- M not compile-time determinable
     C2 : String (1..N);      -- N not compile-time determinable
end record;

type Acc_R is access R;
```

```
function F return Acc_R is
begin ... end F;

package P is
     Obj      : R;
     Ren_C2   : String (1..N) renames F.all.C2;
end P;
```

Now consider the following attributes:

```
P.Obj.C1'Position    -- Easy to compute.
P.Obj.C2'Position    -- Must be computed at run time since C1's
                     -- length is not static.
F.all.C2'Position    -- Same as for P.Obj.C2'Position.
P.Ren_C2'Position    -- Illegal?
```

The last case presents a problem because P.Ren_C2 would normally be implemented as a pointer to the second component of the object containing component C2. But the value of the P.Ren_C2'Position must be calculated using the address of the object containing C2. The code needed to calculate this address cannot usually be generated at the point where P.Ren_C2 is written. For example, one cannot reevaluate F.all in order to determine the object containing Ren_C2. So, because of the possibility of writing P.Ren_C2'Position, the elaboration of the renaming declaration must determine and save the address of F.all (i.e., of the object containing C2). This overhead would be incurred for every renaming of a record component.

It was not the intent to impose such an implementation overhead on renamings of record components. The wording in 13.7.2(7-10) justifies limiting the use of the 'Position, 'First_Bit, and 'Last_Bit attributes to those contexts in which the prefix has the form of a selected component whose prefix denotes a record. The wording in the Annex is not definitive.

!appendix 86-11-19

****

!section 13.07.02 (07) Geoff Mendal/Stanford 86-11-08 83-00860
!version 1983
!topic Error in AI-00258/04

The use of Ren_C2 has a subtle syntax error:

```
  Ren_C2 : String (1 .. N) renames F.all.C2;
```

Renaming an identifier cannot specify a subtype indication, rather only a type mark. LRM 8.5(2).

gom

- - - - End forwarded message - - - -

Ada Commentary ai-00260-bi.wj downloaded on Wed May 11 09:50:04 EDT 1988

!standard 07.04.04 (04) 86-07-23 AI-00260/06
!class binding interpretation 85-08-03
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9-0-1) 85-09-05
!status work-item 85-08-03
!status received 84-06-12
!references 83-00381, 83-00531
!topic Limited "full types"

!summary 85-12-23

A full type declaration declares a limited type if an assignment operation is not visible for the type of some subcomponent at the place of the full type declaration.

A formal parameter whose type mark denotes an incompletely declared private type cannot have mode OUT if the parameter's full type declaration declares a limited type.

!question 85-12-23

The question of when a limited private type is no longer limited affects the legality of declarations with OUT parameters and the legality of full declarations of private types. For example consider the following declarations:

```
-- Example 1

package P is
     type LP1 is limited private;
     type LP2 is limited private;
     type PR is private;
     procedure SET (T : out LP1; S : LP1);   -- illegal?
private
     type PR is
          record
               A : LP2;        -- illegal?
          end record;
     type LP1 is
          record
               B : LP2;
          end record;
     type LP2 is range 1..10;
end P;
```

Is the declaration of SET illegal because "the corresponding full type" for LP1 is limited [7.4.4(4)]? Similarly, is the full declaration of PR illegal because it declares a limited type [see 7.4.1(3)]? Or are both declarations

to be considered legal since equality and assignment are (implicitly) declared for LP1 and PR after LP2's declaration [7.4.2(7)]?

In considering the answer to these questions, it may also be helpful to consider the following example:

```
-- Example 2

package P1 is
     type LP3 is limited private;

     package P2 is
          type LP4 is limited private;
          procedure SET (L : out LP4);  -- illegal?
     private
          type LP4 is array (1..2) of LP3;
     end P2;
private
     type LP3 is range 1..10;
end P1;
```

Is it clear in this case that the declaration of SET is illegal since the assignment and equality operations for LP4 are not declared anywhere within package P1? (They are declared only within P2's package body, which is the "earliest place within the immediate scope of [composite type LP4]" that is after LP3's full type declaration [7.4.2(7)]. Consequently, LP4 is limited throughout P2.)

!recommendation 85-11-04

A type mark denotes a limited type if no assignment operation is visible for it.

A full type declaration declares a limited type if it declares a task type, a type derived from a limited type, or a composite type having a subcomponent for which no assignment operation is visible.

A formal parameter whose type mark denotes an incompletely declared private type cannot have mode OUT if the parameter's full type declaration declares a limited type.

If a full type declaration, F, completes the declaration of one or more previously declared types, then assignment, equality, and inequality operations are implicitly declared for each completed type whose components all have non-limited types. (A component of a completed type has a non-limited type if an assignment operation is visible for the type, either at the point of F's declaration or because an assignment operation is being declared for the type by F's declaration.)

!discussion 85-12-23

7.4.4(1-2) attempt to define the set of limited types. These definitions do

not fully cover the situations exemplified in the question, in which the full declaration of a type does not allow the immediate implicit declaration of assignment and equality (inequality) operations for the type.

The two examples given in the questions might suggest that a full type (the type declared by a full type declaration [7.4.1(3)]) should be considered non-limited as long as assignment and equality operations are eventually declared for the type. This position is not free of difficulties, as is shown by the following example:

```
-- Example 3

package P3 is
     type NP is private;                    -- (0)
     type LP5 is limited private;

     package Q is
          type LARR is array (1..5) of LP5;  -- (1)
     end Q;

private

     type LP5 is new INTEGER;
     type NP is new Q.LARR;                 -- legal? (2)
     X : NP     := ...;                     -- legal  (3)
     Y : Q.LARR := ...;                     -- legal? (4)

end P2;
```

If we take the position that LARR is not limited because assignment and equality operations will be implicitly declared for LARR in Q's package body, then the declaration at (2) must be considered legal since Q.LARR is not a limited type. (Remember that the full declaration of a private type cannot be a limited type.) The initialization at (3) is legal since the assignment operation declared at (0) is visible. But the declaration at (4) is illegal because it does not fall within the scope of the assignment operation for Q.LARR.

The problem with this interpretation of when a type is limited is that the initialization at (4) must be considered illegal even though Q.LARR is not a limited type. It is also strange that initialization is allowed for X's declaration but not for Y's, even though X's type is derived from Y's type.

This example shows that problems arise if 7.4.4(1) is taken too literally. A more consistent view arises if we consider that the properties of a type are determined by what operations are available for a type, i.e., by what operations are declared for a type and the scope of these operations. For example, a type is an integer type if it has operations that allow integer values to be created (implicit conversions in the case of integer types) and other operations that can be applied to these values.

In the case of limited types, it makes sense to consider a type to be limited

if no assignment operation for the type is visible, i.e., if an assignment operation is declared for a type, the type is only considered non-limited throughout the scope of the assignment operation's declaration. This view is consistent with the Standard's view that limited types are distinguished (in part) by the unavailability of assignment operations and by the view that an operation is only usable within the scope of its declaration.

Since the predefined equality and inequality operations are declared at the same time as an assignment operation, it suffices to present rules in terms of the assignment operation. One could equally well speak in terms of the scope of the assignment, equality, and inequality operations.

Some declarations serve to complete the declarations of previously declared private types. For example, the declaration of LP2 completes the declaration of type PR and type LP1 in example 1. The intent of the Standard is that assignment, equality, and inequality operations are implicitly declared after the complete declaration of a type if all components of the completed type are considered non-limited at the point of the complete declaration. For example, the assignment operation for type LP1 is declared after the declaration of type LP2, since LP2's declaration completes the declaration of LP1, LP2 is not (here) a limited type, and LP1's only component has a type that is no longer limited.

Now let us consider the effect of these interpretations on the examples. In example 1, the declaration of type PR is illegal because a component of the record has a limited type, LP2. LP2 is considered to denote a limited type because the scope of LP2's assignment operation does not include the attempted declaration of PR's component. Similarly, the full declaration of LP1 declares a limited type. Hence, the declaration of SET is considered to be illegal because LP1's full declaration declares a limited type. An assignment operation is declared for LP2 since LP2 is an integer type. In addition, since LP2's declaration completes LP1's declaration, each component of LP1 is examined to see if assignment operations have now been declared for each component type. Since an assignment operation has now been declared for LP2 and is visible after LP2's declaration, an assignment operation for LP1 is also declared after LP2's declaration. In P's body, LP1 will not denote a limited type since the scope of the assignment operation's declaration includes P's body. Outside of P, LP1 will be a limited type since the scope of the assignment operation's declaration does not extend outside of P's private part and body.

In example 2, the declaration of SET is similarly illegal because LP4's full declaration declares a limited type -- no assignment operation is visible for component type LP3.

In example 3, NP's declaration is illegal because no assignment operation is visible for type Q.LARR; the assignment operation is declared in package Q's body [7.4.2(7)]. The attempted initialization of Y is also illegal since no assignment operation is visible.

Ada Commentary ai-00261-bi.wj downloaded on Wed May 11 10:45:02 EDT 1988

| !standard 10.03 (03) 86-07-23 AI-00261/03
| !class binding interpretation 84-06-12
| !status approved by WG9/AJPO 86-07-22
| !status approved by Director, AJPO 86-07-22
| !status approved by WG9/Ada Board 86-07-22
| !status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-26
!status work-item 84-06-12
!status received 84-06-12
!references 83-00380, 83-00170
| !topic "any error" -> "any illegal construct"

!summary 85-01-02

An attempt to compile an illegal compilation unit has no effect on the program library (see also AI-00255).

!question 85-01-02

10.3(3) says "If any error is detected while attempting to compile a compilation unit, then the attempted compilation is rejected and it has no effect whatsoever on the program library." The phrase "any error" includes errors that must be detected at run time, erroneous executions, and programs containing incorrect order dependences [1.6(1-10)]. Should 10.3(3) have referred just to those errors detected at compilation time?

!recommendation 85-01-02

If any illegal construct is detected while attempting to compile a compilation unit, the attempt is not successful and has no effect whatsoever on the program library.

!discussion 85-01-02

Since the paragraph is discussing actions taken at compile time (i.e., actions taken prior to execution of the main program), "any error" should be understood to refer just to those errors that must be detected at compilation time, i.e., those errors that make a compilation unit illegal (see [1.6(2,3)]).

The phrase "attempted compilation" refers to the phrase "attempt to compile" and not to the syntactic category "compilation" as defined in 10.1(2) -- "attempted <syntactic category>" makes no sense, and so "compilation" must here be considered to be used in its normal technical sense.

In short, the intent is to say that an attempt to compile an illegal compilation unit has no effect on the program library.

Ada Commentary ai-00265-ra.wj downloaded on Wed May 11 10:45:04 EDT 1988

!standard 04.03.02 (11) 87-06-18 AI-00265/05
!class ramification 84-07-29
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
!status work-item 86-08-12
!status received 84-07-29
!references 83-00394, 83-00394
!topic Index subtype of an array aggregate

!summary 86-09-17

The index subtype of an array type declared by a constrained array definition is the subtype defined by the corresponding discrete range.

!question 86-09-17

Consider the following example:

```
declare
    subtype SUB is INTEGER range 1 .. 3;
    A : array (SUB) of INTEGER;
    B : array (1 .. 3) of INTEGER;
    C : array (INTEGER range 1 .. 3) of INTEGER;
begin
    A := (2=>0, 3=>0, 4=>0);    -- CONSTRAINT_ERROR? (yes)
    B := (2=>0, 3=>0, 4=>0);    -- CONSTRAINT_ERROR? (yes)
    C := (2=>0, 3=>0, 4=>0);    -- CONSTRAINT_ERROR? (yes)
```

Is CONSTRAINT_ERROR raised for any of these cases?

In answering the above question, note 4.3.2(11):

> a check is made that the index VALUES defined by choices belong to the corresponding index subtypes

What is the index subtype for each index of these array aggregates?

!response 86-09-17

For an unconstrained array, 3.6(5) states:

> The index subtype for a given index position is, by definition, the subtype denoted by the type mark of the corresponding index subtype definition.

For a constrained array type definition, 3.6(7) states:

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

Since each of the variables is declared with a constrained array definition, the index subtype for each of the three types is INTEGER range 1..3. Consequently, this is the index subtype for each of the aggregates. Thus, the check stated in 4.3.2(11) fails when the aggregates are evaluated, and CONSTRAINT_ERROR is raised.

Note that no exception would be raised for

```
D : STRING (1 .. 3) := (2 => 'a', 3 => 'b', 4 => 'c');
```

Since STRING is declared as an unconstrained array type, its index subtype is POSITIVE, and 4 belongs to the subtype POSITIVE.

Ada Commentary ai-00266-ra.wj downloaded on Wed May 11 10:50:02 EDT 1988

!standard 10.01 (06) 86-07-23 AI-00266/09
!class ramification 85-02-04
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status WG9/ADA Board approved (provisional) 85-05-13
!status committee-approved (10-0-0) 85-02-27
!status work-item 85-02-04
!status received 84-07-29
!references AI-00199, 83-00389, 83-00520, 83-00578
!topic A body cannot be compiled for a library unit instantiation

!summary 85-03-07

If a generic package is instantiated as a library unit, it is illegal to attempt to compile a package body having the same identifier as that of the instantiation.

After instantiating a generic subprogram as a library unit, any attempt to compile a subprogram body having the same identifier as that of the library unit instantiation causes the instantiation to be deleted from the library and replaced with the new library unit subprogram.

!question 85-02-04

Suppose a generic subprogram or package is instantiated as a compilation unit. What is the effect of compiling a new body with the name and profile of the instantiation? In particular, consider this sequence of compilation units:

```
-- Unit 1:  A library unit that is a generic subprogram
        generic
        procedure GPROC;

-- Unit 2: Its body
        procedure GPROC is
        begin ... end;

-- Unit 3: An instantiation as a library unit
        with GPROC;
        pragma ELABORATE (GPROC);          -- to prevent PROGRAM_ERROR
        procedure NEW_PROC is new GPROC;

-- Unit 4: Attempt to recompile body for NEW_PROC
        procedure NEW_PROC is
        begin ... end;
```

Is Unit 4 legal? If so, what is the effect if GPROC's body is later compiled? Must the parameter and result type profile for Unit 4 be the same as the profile created by Unit 3? Suppose GPROC were a generic package and

Unit 4 was an attempt to compile a new body for the instantiated package. Would Unit 4 be legal?

!response 85-10-23

See the discussion for AI-00199, which covers this situation as well as related situations.

Ada Commentary ai-00267-ra.wj downloaded on Wed May 11 10:50:05 EDT 1988

| !standard 11.06 (06) 86-12-04 AI-00267/04
| !standard 05.05 (07)
| !standard 03.05.04 (10)
| !class ramification 86-04-16
| !status approved by WG9/AJPO 86-11-26
| !status approved by Director, AJPO 86-11-26
| !status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (8-0-1) 86-05-12 (pending editorial review)
!status work-item 86-04-11
!status received 84-07-29
| !references AI-00387, 83-00388
!topic Evaluating expressions in case statements

!summary 86-04-16

An exception is raised if the value of the expression in a case statement does not belong to the base type of the expression.

!question 86-06-25

Assume that there are at least two predefined integer types, as follows:

```
type SHORT_INTEGER is range -256 .. 255;
type INTEGER is range ...;                -- anything longer
```

and consider this code:

```
X : SHORT_INTEGER := SHORT_INTEGER'LAST;  -- 255
...
case (X + 1) is
    when SHORT_INTEGER => null;
end case;
```

11.6(6) says:

> Freedom is left to an implementation for the evaluation of numeric simple expressions. For the evaluation of a predefined operation, an implementation is allowed to use the operation of a type that has a range wider than that of the base type of the operands, provided that this delivers the exact result ... even if some intermediate results lie outside the range of the base type. The exception NUMERIC_ERROR need not be raised in such a case.

The case statement is clearly legal, and (X + 1) is equally clearly a simple expression. Must evaluation of the case statement expression raise NUMERIC_ERROR, or is the case statement expression considered an "intermediate" result for purposes of 11.6?

!response 86-12-04

The value produced by the evaluation of the expression in a case statement is considered not an intermediate result but a "final" value. If the final value does not lie within the range of the base type, an exception must be raised (usually NUMERIC_ERROR, but AI-00387 allows CONSTRAINT_ERROR to be raised instead). 11.6(6) does not apply here.

In the above example, the expression in the case statement must raise NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387).

Ada Commentary ai-00268-ra.wj downloaded on Wed May 11 11:45:02 EDT 1988

!standard 09.03 (03) 86-07-23 AI-00268/06
!class ramification 84-11-06
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-11-06
!status received 84-08-05
!references 83-00383, 83-00411, 83-00430
!topic Activation of already abnormal tasks

!summary 84-11-06

If a task is aborted before it is activated, no exception is raised when an attempt is made to activate the task.

!question 84-11-06

If a task is aborted before it is activated, should an exception be raised when an attempt is made to activate the task?

!response 85-01-04

No rule says a check must be made before activating a task to see if the task is terminated. In the absence such a rule, no such check can be made, and so no exception can be raised if an attempt is made to activate a terminated task.

Although no semantics are given explicitly for an attempt to activate a terminated task, it is reasonable to conclude that only an unactivated task can be activated. An attempt to activate a terminated task simply is unsuccessful; no exception should be raised.

Ada Commentary ai-00276-ra.wj downloaded on Wed May 11 11:45:05 EDT 1988

| !standard 09.07.03 (04) 87-02-23 AI-00276/07
| !standard 09.07.02 (01)
| !class ramification 85-02-27
| !status approved by WG9/AJPO 87-02-20
| !status approved by Director, AJPO 87-02-20
| !status approved by Ada Board (21-0-2) 87-02-19
!status approved by WG9 85-11-18
!status committee-approved (7-1-2) 85-02-27
!status work-item 85-02-04
!status received 84-08-27
!references 83-00406, 83-00437
!topic Rendezvous that are "immediately possible" vs. timed entry calls

!summary 85-09-14

A timed entry call with a zero or negative delay issues an entry call that is canceled only if a rendezvous is not immediately possible. 9.7.2(4) specifies the conditions under which an entry call is immediately possible. In a distributed implementation of Ada, it may take a non-negligible amount of time to determine whether an entry call is "immediately" possible.

!question 85-10-29

9.7.2(1) states, "A conditional entry call issues an entry call that is then canceled if a rendezvous is not immediately possible." Paragraph 4 of the same section then goes on to state the conditions under which such an entry call must be canceled. It is not clear if paragraph 4 is meant to provide a complete definition for the word immediate as used in paragraph 1, or merely to list a necessary set of conditions under which such an entry call must be canceled. While these two statements do not appear inconsistent in the uniprocessor case, the exact interpretation is much less clear when considering the impact on distributed systems.

In distributed environments several interpretations are possible. One interpretation is that due to non-negligible inter-nodal communication delays there can be no "immediate" acceptance of distributed entry calls; hence, non-local conditional entry calls are always canceled. A less restrictive interpretation is to say the determination of immediacy uses the criteria stated in 9.7.2(4) (i.e., the determination of immediacy is completely independent of any communication delays that may be present in the implementation of an arbitrary distributed system). For applications where the absolute elapsed time between a call and accept is important, timed entry calls should be used.

Which interpretation is correct?

!response 85-09-14

Since 9.7.2(1) says an entry call (for a conditional entry call) is canceled "if a rendezvous is not immediately possible," and 9.7.2(4) specifies the conditions under which an entry call is canceled, 9.7.2(4) implicitly defines when an entry call is "not immediately possible". Determining that a called

task is able (and willing) to accept a call might take a long time, especially in a distributed processing environment. Nonetheless, since the definition in 9.7.2(4) does not mention time, "immediately possible" is to be understood as specifying requirements on the state of the called task (and its willingness to accept the call in the case of a selective wait) rather than a requirement that the call be accepted within some small interval of time.

9.7.3(3) says, for a timed entry call:

> If a rendezvous can be started within the specified duration (or immediately, as for a conditional entry call, for a negative or zero delay), it is performed ... Otherwise the entry call is canceled when the specified duration has expired.

This means that if the specified delay is exactly zero or negative, execution of the timed entry call takes as long as needed to decide whether the call can be accepted "immediately," in the sense of 9.7.2(4). If the delay is nonzero and positive, the entry call can be canceled as soon as the delay expires (if it has not already been accepted). If the delay is small enough, it even might be canceled before it has been possible to decide if the call can be accepted "immediately".

Ada Commentary ai-00279-bi.wj downloaded on Wed May 11 11:50:02 EDT 1988

!standard 14.01 (11) 86-12-01 AI-00279/09
!class binding interpretation 86-02-20
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (8-2-0) 86-05-12 (pending editorial review)
!status committee-approved (7-1-0) 86-02-20 (pending editorial review)
!status work-item 85-05-27
!status returned to committee by WG9/ADA Board 85-05-13
!status committee-approved (10-0-0) 85-02-27
!status work-item 85-02-05
!status received 84-08-27
!references AI-00325, AI-00355, 83-00413, 83-00414, 83-00579
!topic Exceptions raised by calls of I/O subprograms

!summary 86-07-07

Any exception can be raised when evaluating the actual parameters of a call of an input-output subprogram. In addition, STORAGE_ERROR can be raised by the call itself before execution of the body has begun. But once execution of the body of an input-output subprogram has been started, the only exceptions that can be propagated to the caller are the exceptions defined in the package IO_EXCEPTIONS and the exceptions PROGRAM_ERROR and STORAGE_ERROR. In particular, if CONSTRAINT_ERROR, NUMERIC_ERROR, or TASKING_ERROR is not raised by the evaluation of any argument, then none of these exceptions will be raised by the call. Furthermore, PROGRAM_ERROR can only be raised due to errors made by the user of the input-output subprogram.

!question 86-07-02

14.1(11) states:

> The exceptions that can be raised by a call of an input-output subprogram are all defined in the package IO_EXCEPTIONS; the situations in which they can be raised are described, either following the description of the subprogram (and in section 14.4), or in Appendix F in the case of error situations that are implementation-dependent.

Despite this wording, it appears that other exceptions can be raised by a call of at least some input-output subprograms, e.g.,

```
NEW_LINE(-1);
```

will raise CONSTRAINT_ERROR because the argument does not belong to the formal parameter's subtype. NEW_LINE(A*B) might raise NUMERIC_ERROR when A*B is evaluated. PROGRAM_ERROR could be raised by NEW_LINE(C) if C has an undefined value or if the body of NEW_LINE has not been elaborated at the time of the call. TASKING_ERROR or some user-defined exception might be raised when evaluating a function appearing as an actual parameter. STORAGE_ERROR could be raised if there is insufficient space left to allow

the subprogram to be called. In short, despite the wording of 14.1(11), it appears that any exception can be raised by a call of at least some I/O subprograms. Was it the intent just to limit the exceptions that can be raised from within the body of a subprogram? For example, was it the intent to forbid a call of GET from raising NUMERIC_ERROR when reading the following literals:

```
9999999999999999999999999999999999999999999999990
12E+9999999999999999999999999999999999999999999990
```

!recommendation 86-07-02

After execution of the body of an input-output subprogram has started, the only exceptions that can be propagated to the caller are the exceptions defined in the package IO_EXCEPTIONS and the exceptions PROGRAM_ERROR and STORAGE_ERROR.

!discussion 86-08-05

14.1(11) limits the exceptions that can be propagated from the body of an input-output subprogram. As is noted in the question, evaluation of a call's actual parameters can, in general, cause any exception to be raised. However, once execution of the body has started, only those exceptions defined in the IO_EXCEPTIONS package and the exceptions PROGRAM_ERROR and STORAGE_ERROR are allowed to be propagated.

STORAGE_ERROR can be raised after the actual parameters have been evaluated and before execution of the body has been started, as well as during the execution of the body if storage is not sufficient. (Of course, it is expected that implementations make efficient use of storage so STORAGE_ERROR is not raised unnecessarily; see AI-00325).

PROGRAM_ERROR can be raised upon an attempt to execute an action that is erroneous and that the implementation cannot reasonably defend against. For example, the following example illustrates an attempt to access an actual parameter that has not been properly initialized (i.e., string with an undefined component):

```
with TEXT_IO; use TEXT_IO;
procedure ... is
    STR : STRING (1 .. 80);
begin
    STR(1 .. 5) := "STARS";    -- STR(6 .. 80) is uninitialized
    PUT (STR);                 -- may raise PROGRAM_ERROR
```

PROGRAM_ERROR can also be raised if the body of the input-output subprogram has not yet been elaborated, but this can be avoided with a properly placed pragma ELABORATE (see AI-00355).

With respect to the example given in the question, a call, GET(ITEM), raises DATA_ERROR after reading stops (not NUMERIC_ERROR or CONSTRAINT_ERROR), assuming the first two values exceed NUM'LAST (NUM is the type used in the

instantiation of INTEGER_IO). In both of the cases given in the question, reading stops at the end of the literal, i.e., after the zero, since reading is "according to the syntax" of an integer or real literal (14.3.7(6)). The call of GET should never raise NUMERIC_ERROR.

Ada Commentary ai-00282-co.wj downloaded on Wed May 11 11:50:03 EDT 1988

!standard 03.06.01 (04) 86-12-01 AI-00282/06
!class confirmation 86-05-12
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (10-0-0) 86-05-12 (pending editorial review)
!status committee-approved (7-0-1) 86-02-20 (pending editorial review)
!status work-item 86-01-24
!status received 84-09-28
!references 83-00417
!topic Compatibility of constraint defined by discrete range

!summary 86-06-24

The constraint defined by a discrete range is compatible with a subtype if each bound of the discrete range belongs to the subtype, or if the discrete range defines a null range; otherwise the constraint is not compatible with the subtype.

!question 86-06-27

3.6.1(4) says:

> An index constraint is compatible with the type denoted by the type mark if and only if the constraint defined by each discrete range is compatible with the corresponding index subtype.

The Standard does not, however, explicitly define what it means for a discrete range to be compatible with a subtype. The closest thing to a definition comes in 3.5(4):

> A range constraint is compatible with a subtype if each bound of the range belongs to the subtype, or if the range constraint defines a null range; otherwise the range constraint is not compatible with the subtype.

Since a discrete range is not, syntactically, a range constraint, the definition in 3.5(4) does not, strictly speaking, define compatibility of a discrete range with an index subtype.

Should compatibility of a discrete range with an index subtype be defined in essentially the same way as compatibility of a range constraint?

!response 86-06-27

There is no gap in the language if we consider the following points:

a) The notion of compatibility is defined for constraints, 3.3.2(9).

b) The "missing" notion is the compatibility of:

> a constraint defined by the discrete range with the corresponding index subtype,

and not the compatibility of

> a discrete range with the corresponding index subtype.

The above "missing" notion is due to a misunderstanding of the definition of "a constraint defined by a discrete range." If the word "defined" is used in the ordinary English sense of "specified" or "determined," then we realize that the compatibility is of:

> a constraint that is specified (or determined) by the bounds of the discrete range.

Hence, a discrete range defines a constraint in the same way as a range constraint (considered as a syntactic term) defines a constraint.

Ada Commentary ai-00286-bi.wj downloaded on Wed May 11 12:45:01 EDT 1988

87-06-18 AI-00286/11

!standard 12.01 (05)
!standard 08.04 (05)
!standard 08.03 (15)
!class binding interpretation 85-02-05
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-11-14 (reviewed)
!status 86-09-13 (pending editorial review)
!status 86-08-07 (pending editorial review)
!status committee-approved (8-2-0) 86-05-12 (pending editorial review)
!status committee-approved (6-1-1) 86-02-21 (pending editorial review)
!status work-item 85-02-05
!status received 84-10-01
!references 83-00423, 83-00435, 83-00486, 83-00488, 83-00584, 83-00713, 83-00714, 83-00715, 83-00721
!topic Declarations visible in a generic subprogram decl and body

!summary 86-10-02

Except within the body of a generic subprogram, the declaration of a generic unit is not a declaration for which overloading is allowed. In particular, any declarations occurring in an outer declarative region or made potentially visible by a use clause are not directly visible in the generic formal part if they have the same identifier as the subprogram.

Within the body of a generic subprogram, overloading is defined for the generic subprogram declaration in the same way as for a nongeneric subprogram. In particular, overloadable declarations occurring in an outer declarative region or made potentially visible by a use clause can be directly visible in the body even though they are not directly visible in the generic formal part.

!question 86-04-09

Consider the following declarations:

```
package PACK is
     function F return BOOLEAN;
end PACK;

with PACK; use PACK;
procedure P is
     function F return INTEGER is ... end F;

     procedure INNER is
          generic
               W1 : FLOAT   := F;     -- illegal; 8.3(5)
               X1 : INTEGER := F;     -- legal? (no)
               Y1 : BOOLEAN := F;     -- legal? (no)
```

```
                    Z1 : INTEGER := P.F;  -- legal
               function F return FLOAT;

               function F return FLOAT is
                    W2 : FLOAT   := F;    -- legal (recursive call)
                    X2 : INTEGER := F;    -- legal? (yes)
                    Y2 : BOOLEAN := F;    -- legal? (yes)
                    Z2 : INTEGER := P.F;  -- legal
               begin ... end F;
          begin ... end INNER;
     begin
          ...
     end P;
```

If generic function F is considered overloadable within its generic formal part, then both the outer F and PACK.F are directly visible within the generic formal part and the attempted initializations of X1 and Y1 are legal. If generic function F is not overloadable, then no F outside the generic formal part is directly visible. Are the declarations of P.F and PACK.F visible (directly or by selection) within F's generic formal part? within F's body?

!recommendation 86-10-02

Except within the body of a generic subprogram, the declaration of a generic unit is not a declaration for which overloading is allowed. Within the subprogram body, overloading is defined for the generic subprogram declaration as for a nongeneric subprogram declaration.

!discussion 87-06-04

12.1(5) says:

> Outside the specification and body of a generic unit, the name of this program unit denotes the generic unit. In contrast, within the declarative region associated with a generic subprogram, the name of this program unit denotes the subprogram obtained by the current instantiation of the generic unit.

A note, 12.1(10), explains the intent of 12.1(5):

> Within a generic subprogram, the name of this program unit acts as the name of a subprogram. Hence this name can be overloaded, and it can appear in a recursive call of the current instantiation.

Now let us consider the example given in the question. First, the attempted initialization of W1 is illegal because generic function F is not visible within its own generic formal part (8.3(5)). Next, since the declarative region of a generic unit includes its generic formal part (8.1(2)), and since 12.1(5, 10) imply that the name of a generic subprogram can be overloaded

within its declarative region, one might conclude that within the generic formal part and body of F, both P.F and PACK.F are directly visible. Unfortunately, if we take this interpretation, an example can be constructed in which it is impossible to decide whether an outer declaration is visible within a generic formal part:

```
function G return INTEGER is                              -- G1
     subtype T is INTEGER;

     function G return BOOLEAN is                         -- G2
          subtype T is BOOLEAN;

          function H return INTEGER is                    -- H1
               subtype U is INTEGER;

               function H return BOOLEAN is               -- H2
                    subtype U is BOOLEAN;

                    generic
                         with function G return H.U;      -- G3: hides
                                                          -- either G1
                                                          -- or G2.
                    function H return G.T;                -- H3

               end H;
          end H;
     end G;
end G;
```

Consider the visibility of G in the prefix G.T. G1, G2, and G3 are all potentially visible, but to determine the visibility of G1 and G2, we need to know the parameter and result type profile for G3, since if G3 is a homograph of G1 or G2, these declarations would be hidden. To determine this profile we need to know what H.U denotes, so we need to know what Hs are visible. H1, H2, and H3 are all potentially visible, but H3 could hide either H1 or H2. To decide whether H3 hides either of H1 or H2, we need to know what G.T denotes, and this means deciding what Gs are visible. But this is where we started the analysis. So there is a loop. The same loop occurs if we start the analysis with the prefix of H.U.

It should be noted that the example is not illegal because there is more than one visible enclosing G and H (which would make G.T and H.U illegal expanded names; see 4.1.3(18)). If G.T is replaced with INTEGER, then H.U unambiguously denotes BOOLEAN, since H1 is hidden by H3, and H.U means H2.U (i.e., there is only one VISIBLE enclosing subprogram named H). Similarly, if G.T is replaced with BOOLEAN, H.U denotes INTEGER. Similar analysis works for replacing H.U with either INTEGER or BOOLEAN.

In short, the existence of the circularity means we must interpret the Standard in a way that removes the problem.

The difficulty is resolved if a generic subprogram cannot be overloaded,

either within or outside the generic unit. If overloading is not allowed, H.U is illegal since H can only denote H3, and no U is declared in the formal part. Similarly, with respect to the example given in the question, the initializations of X1 and Y1 would be illegal since P.F and PACK.F would not be directly visible within the generic formal part. In addition, the initializations of X2 and Y2 would be illegal.

This interpretation might seem to be justified by the first sentence of 12.1(5), but the second sentence implies that a generic subprogram can be overloaded at least within its body, i.e., new overloadable declarations having the generic unit's identifier can be given within the body without hiding the subprogram denoted by the generic unit's identifier. One might argue that although overloading is allowed within the body, no outer declaration having the generic unit's identifier is directly visible. For example, we could rewrite the body of generic unit F as:

```
function F return FLOAT is
    function F return INTEGER renames P.F;
    use PACK;
    W2 : FLOAT   := F;      -- P.INNER.F; recursive call
    X2 : INTEGER := F;      -- denotes renamed F
    Y2 : BOOLEAN := F;      -- illegal - not directly visible
```

The renaming declaration does not hide the generic unit's declaration since the generic unit's identifier is considered to denote a subprogram, an overloadable entity. The use clause does not make PACK.F directly visible if overloading is not allowed for generic function F outside its body.

An alternative resolution is to consider the generic unit's name to be fully overloadable within the subprogram body, i.e., within the body, both P.F and PACK.F are directly visible; within the generic formal part, P.F and PACK.F are not directly visible. Although it may at first seem inconsistent for entities to be visible in the body but not in the generic formal part when both places are part of the same declarative region, this change of visibility between specification and body already exists for non-generic subprograms. In particular, suppose we replace the declaration of generic unit F with the following non-generic declaration:

```
function F (WW1 : FLOAT   := F;      -- illegal
            XX1 : INTEGER := F;      -- illegal
            YY1 : BOOLEAN := F;      -- illegal
            ZZ1 : INTEGER := P.F     -- illegal
           ) return FLOAT;

function F (WW1 : FLOAT   := ...;
            XX1 : INTEGER := ...;
            YY1 : BOOLEAN := ...;
            ZZ1 : INTEGER := ...
           ) return FLOAT is
    WW2 : FLOAT   := F;              -- legal
    XX2 : INTEGER := F;              -- legal
    YY2 : BOOLEAN := F;              -- legal
```

8.3(16) says that within the specification of a subprogram, no declarations having the same designator as the subprogram are visible, either directly or by selection. Hence, neither P.F nor PACK.F can be referenced within the F's specification (even as expanded names). However, within the body of F, 8.3(16) does not apply, so P.F and PACK.F are directly visible. The recommended interpretation is consistent with this example, which shows that an outer declaration can be hidden at one point in a declarative region and yet be directly visible later in the same declarative region.

One might argue that the reasons underlying 8.3(16) apply equally well to the generic formal part and so the recommendation should extend 8.3(16) to include the generic formal part of a generic subprogram's declaration. Such an extension would imply that declarations having the same designator as the subprogram are not visible either directly or by selection within the generic formal part (making illegal the use of P.F in Z1's initialization). But a key difference between a generic unit declaration and a non-generic subprogram declaration is that within the declarative region that immediately contains the generic subprogram declaration there can be no other explicitly declared subprogram having the same identifier; overloading is not allowed. A non-generic subprogram can be overloaded, of course. This difference means some of the motivation behind the rule in 8.3(16) (see AI-00370) does not apply to generic units. Moreover, it is unnecessary to extend the 8.3(16) rule to generic formal parts to resolve the problem posed by the example with functions G and H. It is sufficient (and preferable) to restrict only direct visibility of outer declarations.

Ada Commentary ai-00287-bi.wj downloaded on Wed May 11 12:45:03 EDT 1988

!standard 09.05 (05) 86-07-23 AI-00287/05
!standard 08.07 (13)
!class binding interpretation 84-10-01
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (7-0-1) 86-02-20
!status work-item 86-01-24
!status received 84-10-01
!references 83-00426
!topic Resolving overloaded entry calls

!summary 86-02-27

A call to an overloaded entry is resolved using the same kind of information as is used for resolving overloaded procedure calls (the name of the entry, the number of parameters, the types and the order of the actual parameters, and the names of the formal parameters).

!question 86-02-27

No rule is given for entry calls that is analogous to the rule given in 6.6(3), which states what information is used in resolving overloaded subprogram calls. Does the resolution of entry calls use the same information as is used for resolving procedure calls?

!recommendation 86-01-24

A call to an overloaded entry is ambiguous (and therefore illegal) if the name of the entry, the number of parameter associations, the types and the order of the actual parameters, and the names of the formal parameters (if named associations are used) are not sufficient to determine exactly one (overloaded) entry specification.

When considering possible interpretations of a complete context, the above rule for entry calls is taken into consideration.

!discussion 86-01-24

The resolution of overloaded entry calls was intended to follow the same rules as the resolution of subprogram calls. In particular, the mode of the parameters was not to be considered. It was an oversight that an explicit rule to this effect was not given in the Standard, and moreover, that 8.7(13) does not mention that such a rule is used when resolving overloaded entry calls.

Ada Commentary ai-00288-nb.wj downloaded on Wed May 11 12:50:01 EDT 1988

!standard 09.08 (04) 86-07-23 AI-00288/06
!class non-binding interpretation 85-11-22
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9-1-2) 86-02-20
!status committee-approved (8-1-0) 85-11-22 (pending letter ballot)
!status work-item 85-04-10
!status received 84-10-01
!references 83-00424
!topic Effect of priorities during activation

!summary 85-12-27

A task activation should be performed with the priority of the task being activated or the priority of the task causing the activation, whichever is higher.

!question 86-01-20

Suppose a high priority task activates a low priority task at a time when a task of intermediate priority is eligible for execution. The high priority task cannot execute until the low priority task is activated, but the low priority task cannot be executed (and thereby activated) until the intermediate priority task is no longer eligible for execution. In short, since task activation is performed at the priority of the task being activated, execution of the activating task may be delayed until the task being activated has the highest priority of those eligible to run. This behavior seems inconsistent with the behavior during a rendezvous, since a rendezvous executed on behalf of a high priority task is executed at the priority of that task or higher.

Was it intended that activations be performed at the priority of the task causing the activation, if this task's priority is higher?

!recommendation 85-12-16

When a task causes the activation of another task, if the priorities of both tasks are defined, the activation is executed with the higher of the two priorities. If only one of the two priorities is defined, the activation is executed with at least that priority. If neither is defined, the priority of the activation is undefined.

!discussion 85-12-27

Task activation is similar to rendezvous in that one task waits until another completes a requested action. In the case of a rendezvous, the calling task must wait until the rendezvous is complete (i.e., until execution of the accept statement is complete). In the case of activation, the task causing the activation must wait until the task being activated has elaborated its

declarative part. In both cases, it is undesirable for the execution of high priority tasks to be unnecessarily delayed. This goal is accomplished for rendezvous by rules (9.8(5)) saying that a rendezvous is executed with the priority of the calling or called task, whichever is higher. A similar rule should have been given for task activation -- when a task causes the activation of other tasks, the activation should be performed with either the priority of the task causing the activation or the priority of the task being activated, whichever is higher.

Ada Commentary ai-00289-bi.wj downloaded on Wed May 11 12:50:03 EDT 1988

!standard 10.02 (05) 86-07-23 AI-00289/05
!class binding interpretation 86-01-28
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (5-2-1) 86-02-20
!status work-item 86-01-22
!status received 84-10-01
!references 83-00432
!topic Ancestor unit names in separate clauses must be simple names

!summary 86-02-27

If P is a library unit, then the name in a "separate" clause for a subunit of P must be P and not STANDARD.P.

!question 86-01-28

10.2(5) says:

> Each subunit mentions the name of its parent unit, that is, the compilation unit where the corresponding body stub is given. If the parent unit is a library unit, it is called the ancestor library unit. If the parent unit is itself a subunit, the parent unit name must be given in full as an expanded name, starting with the simple name of the ancestor library unit.

This means that if the parent unit's name is P.Q, it may not be given as STANDARD.P.Q in a "separate" clause; but if the parent unit's name is P, i.e., if the parent unit is not a subunit but is in fact a library unit, there seems to be no rule forbidding the use of STANDARD.P. Can a compiler allow names of the form STANDARD.P in "separate" clauses? Must it?

!recommendation 86-02-27

If the parent unit of a subunit is not a subunit, the parent unit name in the "separate" clause for the subunit must be given as a simple name.

!discussion 86-01-28

10.1(4) says:

> The effect of compiling a library unit is to define (or redefine) this unit as one that belongs to the program library. For the visibility rules, each library unit acts as a declaration that occurs immediately within the package STANDARD.

This means that one can usually write STANDARD.P to refer to library unit P.

The rule in 10.2(5) only specifies the form of parent unit name when the

parent unit is a subunit, forbidding names of the form STANDARD.P.Q when P is the ancestor library unit of a subunit. Since this form of name is forbidden when a parent unit is a subunit (and hence, not a library unit), clearly the intent was to forbid it also when the parent unit is a library unit (or the secondary unit of a library unit).

Ada Commentary ai-00292-bi.wj downloaded on Wed May 11 13:45:02 EDT 1988

!standard 03.04 (10) 86-12-01 AI-00292/05
!standard 13.05 (08)
!class binding interpretation 84-10-16
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (9-0-0) 85-11-22 (pending editorial review)
!status received 84-10-16
!references AI-00138, AI-00379, 83-00448
!topic Derived types with address clauses for entries

!summary 86-04-26

An address clause applied to an entry of a task type also applies to a type derived (directly or indirectly) from the task type.

!question 86-01-06

Consider the following example:

```
task type T is
     entry E;
     for E use ...;
end T;
type D is new T;
X : D;
```

Does the address clause of type T apply to the derived type D?

!recommendation 86-04-30

If an address clause has been given for an entry of a task type and the task type is used as a parent type in a derived type declaration, the same address clause is given for the corresponding entry of the derived type.

!discussion 86-01-07

This is a special case of AI-00138. Since T is the parent type of D and the address clause specifies an aspect of the parent task type entry (T.E), then the entry for D and the entry for T have the same address.

AI-00379 discusses the meaning of a program that has two task objects that have entries associated with the same interrupt.

Ada Commentary ai-00293-co.wj downloaded on Wed May 11 13:45:06 EDT 1988

!standard 04.03 (06) 87-06-18 AI-00293/05
!class confirmation 84-10-16
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (22-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
!status work-item 86-04-04
!status received 84-10-16
!references 83-00458
!topic Null others choice for array aggregates

!summary 86-09-12

The others choice in an array aggregate can specify no components.

!question 86-09-15

4.3.2(3) says:

> A named association of an array aggregate is only allowed to have a choice that is not static, or likewise a choice that is a null range, if the aggregate includes a single component association and this component association has a single choice.

Can an association with the choice 'others' specify no components since it is not a range and hence not a "null range"? For example:

```
type T is (RED, BLUE, GREEN);
type A is array(T) of BOOLEAN;

function FUNC return A is
begin
    return (T => TRUE, others => FALSE);    -- legal? (yes)
end FUNC;
```

Is the others in the aggregate legal?

!response 87-02-23

The others choice in an array aggregate can specify no components. 4.3(5) states:

> ... The choice others is only allowed in a component association if the association appears last and has this single choice; it specifies all remaining components, if any.

The "if any" implies an others choice is acceptable even if it specifies no components, so the use of others in the aggregate is legal.

Ada Commentary ai-00294-ra.wj downloaded on Wed May 11 13:50:02 EDT 1988

!standard 04.08 (11) 87-08-20 AI-00294/05
!class ramification 87-06-04
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
!status work-item 86-04-14
!status received 84-10-16
!references 83-00452
!topic The name given in pragma CONTROLLED

!summary 87-08-20

The type name given in a pragma CONTROLLED cannot be declared by a subtype declaration nor can it be a first named subtype since a derived type is not allowed.

!question 86-08-01

4.8(11) states:

> A pragma CONTROLLED for a given access type is allowed at the same places as a representation clause for the type (see 13.1). This pragma is not allowed for a derived type.

This is clear, but in 13.1(12) we find for pragma PACK:

> The position of a PACK pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause ...

Presumably, it was the intent to have the same restrictions on the named type for pragma CONTROLLED as for pragma PACK. Although the italicized syntax for pragma CONTROLLED specifies a type name, so does that for pragma PACK, and the pragma PACK allows a first named subtype as well. Is this correct?

In particular, is the following not allowed?

```
type T is access ...
subtype S is T;
pragma CONTROLLED (S);    -- allowed? (no)
```

!response 86-08-01

The restriction on the named type for the pragma PACK is that it applies to a type or to a first named subtype (that is, to a subtype declared by a type declaration, the base type being therefore anonymous). For the pragma CONTROLLED, a first named subtype is not allowed because 4.8(11) says that the pragma is not allowed for a derived type. Since there is no way to generate a first named subtype for an access type except by a derived type declaration, first named subtypes are automatically excluded.

A subtype declaration declares a subtype (3.3.2(1)). A type declaration declares a type (3.3.1(1)) and sometimes a first named subtype as well. Although a type is a subtype of itself (3.3(4)), only a type declaration declares a type. Therefore, a name declared by a subtype declaration is not allowed for pragma CONTROLLED.

Of course, use of a subtype name in the pragma does not make the program illegal, but instead, means that the pragma has no effect (2.8(9)).

Ada Commentary ai-00298-ra.wj downloaded on Wed May 11 13:50:03 EDT 1988

!standard 10.05 (04) 86-12-01 AI-00298/05
!standard 13.09 (03)
!class ramification 84-10-16
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-09-11 (reviewed)
!status panel/committee-approved 86-08-07 (pending editorial review)
!status committee-approved (9-0-0) 86-05-12 (pending editorial review)
!status work-item 86-04-14
!status received 84-10-16
!references AI-00306, AI-00180, 83-00457
!topic Interaction between pragmas ELABORATE and INTERFACE

!summary 86-04-14

A pragma ELABORATE can be applied to a library unit whose body is supplied by a pragma INTERFACE.

!question 86-04-16

Assume these two library units are compiled in the given order:

```
procedure P;
pragma INTERFACE (SOME_LANGUAGE, P);
----------------
with P;
pragma ELABORATE (P);
procedure Q is...
```

According to 10.5(4), the library unit mentioned in a pragma ELABORATE "must have a library unit body". By 13.9(3), a body is "not allowed" for P above, so the second compilation is illegal. Is this correct?

!response 86-08-05

The name given in a pragma ELABORATE can never make a compilation unit illegal, because if a pragma's argument does not correspond to what is allowed for the pragma, the pragma has no effect (2.8(9)). Therefore, if a pragma INTERFACE has been specified for a library subprogram and accepted by an implementation, it is not illegal to name this subprogram in a pragma ELABORATE.

It is not necessary to give a pragma ELABORATE for such a subprogram, since an elaboration check need not be performed (see AI-00180). If an implementation nonetheless does perform an elaboration check, then it is consistent with AI-00180 for the implementation to also specify if the pragma ELABORATE has any useful effect for such subprograms.

Ada Commentary ai-00300-bi.wj downloaded on Wed May 11 14:45:01 EDT 1988

!standard 13.02 (02) 87-06-18 AI-00300/07
!class binding interpretation 84-10-16
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved (11-1-1) 86-11 (by ballot)
!status panel/committee-approved (4-1-0) 86-09-11 (pending letter ballot)
!status work-item 86-07-24
!status received 84-10-16
!references 83-00453, 83-00636
!topic Prefixes of attributes in length clauses

!summary 86-09-17

The prefix of the attribute that appears in a length clause must be a simple name. An expanded name, or the name T'BASE, is not allowed.

!question 86-09-17

What is permissible as the prefix of an attribute in a length clause? Consider these forms:

```
for T'SIZE use N;              -- (1)

for STANDARD.P.T'SIZE use N;   -- (2)

for T'BASE'SIZE use N;         -- (3)
```

Clearly (1) is allowed and, since (2) is harmless, it is presumably allowed, but is (3) intended to be allowed? Similarly, are length clauses with T'BASE'SMALL illegal? Are length clauses with T'BASE'STORAGE_SIZE legal (assuming T is a task type or non-derived access type) and synonymous with the similar length clause using T'STORAGE_SIZE?

!recommendation 86-09-17

The prefix of the attribute that appears in a length clause must be a simple name.

!discussion 86-09-17

The syntax for length clauses in 13.2(2) places no restriction on the form of prefix that can occur in the attribute. It thus appears that examples such as the following are allowed:

```
procedure PROC is

    type T is ...;
    for PROC.T'SIZE use ...;

begin ... end;
```

However, since 13.1(5) states:

> A representation clause and the declaration of the entity to which the clause applies must both occur immediately within the same declarative part, package specification, or task specification; ...

it follows that any prefix in the prefix (e.g., PROC in the above example) can only be either the attribute T'BASE or an identifier or selected component that denotes the immediately containing unit. Allowing a selected component in the prefix of the attribute of a length clause:

a) adds no expressive power (i.e., adds no functionality),
b) is unlike other representation clauses and related pragmas, and
c) does complicate the implementation.

This was an unintended generality that resulted from the (convenient) use of the syntactic term attribute in the description of the length attribute. It was not intended to allow T'BASE as a prefix of the attribute that appears in a length clause, nor is it necessary to allow an expanded name as a prefix. The intent was to allow only a simple name as the prefix.

Ada Commentary ai-00307-ra.wj downloaded on Wed May 11 14:45:02 EDT 1988

| !standard 14.03.07 (06) 86-07-23 AI-00307/04
!standard 14.03.07 (14)
!standard 14.03.08 (09)
!standard 14.03.08 (18)
!standard 14.03.09 (06)
!standard 14.03.09 (11)
!class ramification 84-11-28
| !status approved by WG9/AJPO 86-07-22
| !status approved by Director, AJPO 86-07-22
| !status approved by WG9/Ada Board 86-07-22
| !status approved by Ada Board 86-07-22
| !status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-11-06
!status received 84-10-16
!references 83-00420, 83-00436
!topic GET at end of file and from a null string

!summary 85-01-04

END_ERROR is raised (not DATA_ERROR) when attempting to read integer, real, or enumeration values from a string that is null or that only contains blanks.

END_ERROR (not DATA_ERROR) is raised when attempting to read integer, real, or enumeration values from a file that has no remaining elements or whose only remaining elements are blanks, line terminators, or page terminators.

!question 84-11-06

If a GET is attempted from a null string for integer, real, or enumeration types, should DATA_ERROR be raised or END_ERROR?

If such a GET is attempted when positioned just before a file terminator, should DATA_ERROR be raised or END_ERROR?

!response 85-01-26

14.3.7(6) says for GET:

> "If the value of the parameter WIDTH is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus or minus sign if present, then reads according to the syntax of an integer literal."

Suppose when GET is called to read an integer value, the only remaining characters in a file are blanks, line terminators, and page terminators. Is END_ERROR raised in such a case, i.e., is an attempt made to skip a file terminator?

When WIDTH is zero, the only condition that allows reading to stop is encountering a sequence of characters that does not satisfy the rules for a

legal integer literal. In this case, since a file terminator is not a character, an attempt is made to read beyond the file terminator, and END_ERROR is raised.

When a GET is attempted from a string that is null or that contains blanks, should END_ERROR or DATA_ERROR be raised? 14.3.7(14) says the end of the string is treated as a file terminator and that GET "[follows] the same rules as the GET procedure that reads an integer value from a file." Since END_ERROR is raised when reading from a file that is empty or that contains just blanks, END_ERROR should also be raised when reading from a string that is empty or full of blanks.

The advantage of raising END_ERROR rather than DATA_ERROR in these situations is that the two exceptions distinguish between reading no value and reading something that is not a valid value.

The current tests require that END_ERROR be raised in these situations.

Ada Commentary ai-00308-bi.wj downloaded on Wed May 11 14:50:01 EDT 1988

!standard 03.02.01 (16) 86-07-23 AI-00308/05
!class binding interpretation 84-10-31
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved (9-0-0) 85-11-22
!status work-item 85-02-05
!status received 84-10-31
!references 83-00460
!topic Checking default initialization of discriminants for compatibility

!summary 85-02-05

When an object of a type with discriminants is created either by an object declaration or an allocator, and the values of the object's discriminants are determined by default, each discriminant value is checked for compatibility, as defined in 3.7.2(5). CONSTRAINT_ERROR is raised if this check fails.

!question 85-02-05

Consider the following example:

```
type R (D : INTEGER := -1) is
     record
          COMP : STRING (D .. 10);
     end record;

X : R;                -- CONSTRAINT_ERROR?
Y : R(-1);            -- CONSTRAINT_ERROR
```

According to the current wording in 3.2.1(6, 13, 15, and 16), when X's declaration is elaborated, the default expression for D is evaluated and the default initial value for D is checked to see if it "belongs to the subtype" of D. The value -1 does belong to the subtype INTEGER, so no exception is raised. When an explicit discriminant constraint is given with the same value, however, as in Y's declaration, CONSTRAINT_ERROR is raised because -1 is not compatible with its use in constraining component COMP (3.7.2(5)).

It would seem more consistent for the declaration of X to raise CONSTRAINT_ERROR as well. If no exception is raised, then it is unclear what should happen when evaluating a name such as X.COMP(-1).

!recommendation 85-02-05

When a discriminant is initialized, its value is checked for compatibility.

!discussion 85-02-05

It was an oversight that the Standard specifies that a discriminant value should merely be checked that it belongs to the subtype of the discriminant. Such values should also be checked for compatibility.

Ada Commentary ai-00310-bi.wj downloaded on Wed May 11 14:50:02 EDT 1988

!standard 04.03.02 (03) 86-07-23 AI-00310/04
!class binding interpretation 84-10-31
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-26
!status work-item 84-11-01
!status received 84-10-31
!references 83-00461
!topic OTHERS choices and static index constraints

!summary 84-11-01

An others choice is static if the corresponding index subtype is static and if the corresponding index bounds were specified with a static discrete range in the applicable index constraint.

!question 84-12-10

If an aggregate has more than one component association, and the last component association has an others choice, then the others choice must be static. 4.3.2(3) defines a static others choice as follows: "An OTHERS choice is static if the applicable index constraint is static." Now consider the following example (drawn from test B43201B-B of Version 1.5):

```
N : INTEGER := 3;
subtype NON_STATIC is INTEGER range 1..N;
type ARR is array (NON_STATIC, 1..3) of INTEGER;
...

ARR'(2      => (1..3   => 2),
     others => (1..3   => 3))    -- illegal

ARR'(1..3   => (2      => 2,
                others => 3))    -- legal?
```

Both aggregates require static others choices. The first aggregate is clearly illegal; the corresponding index constraint is non-static. Is the second aggregate also illegal?

!recommendation 84-11-01

An others choice is static if the corresponding index subtype is static and if the corresponding index bounds were specified with a static discrete range in the applicable index constraint.

!discussion 84-10-31

One might argue that the second aggregate is illegal because the corresponding index constraint is (NON_STATIC, 1..3), and this index

constraint is non-static. (4.9(11) says "a static index constraint is an index constraint for which each index subtype of the corresponding array type is static and in which each discrete range is static.") Since the term "index constraint" refers to a syntax rule [1.5(6)], there is no one-dimensional corresponding index constraint that could be considered non-static.

On the other hand, one might argue that the second aggregate is legal because 4.3.2(2) says "the rules concerning array aggregates are formulated in terms of one-dimensional aggregates," implying that as long as the corresponding dimension has a static index subtype and has bounds specified with a static discrete range, an others choice is considered to be static.

Either interpretation is easy to implement. The second interpretation (which makes the second aggregate legal) is also more intuitive.

The ACVC test in Version 1.5 says the second aggregate is legal, but this test was protested by an implementer who pointed out the first argument. It has since been argued that the test correctly reflects the intent of the Standard and should not be withdrawn from release 1.5 or changed in release 1.6.

Since the intent was to consider the staticness of index bounds for each dimension separately, an others choice should be considered static as long as the corresponding index bounds of the applicable index constraint were specified with a static discrete range; the corresponding index subtype must also be static. This interpretation makes the second aggregate legal and means the Version 1.5 test is correct.

Ada Commentary ai-00311-bi.wj downloaded on Wed May 11 15:45:02 EDT 1988

!standard 11.01 (06) 86-07-23 AI-00311/06
!class binding interpretation 84-10-31
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-05-13
!status committee-approved (9-2-1) 85-05 (by letter ballot)
!status committee-approved (5-2-1) 85-02-25 (pending letter ballot)
!status work-item 84-11-01
!status received 84-10-31
!references AI-00325, 83-00462
!topic No NUMERIC_ERROR for null strings

!summary 85-03-11

When computing the upper bound of a null string literal, NUMERIC_ERROR must not be raised, even if the lower bound has no predecessor (but see AI-00325).

!question 85-05-27

11.1(6) says that NUMERIC_ERROR can be raised when an implementation "uses a predefined numeric operation for the execution, evaluation, or elaboration of some construct," and the operation "cannot deliver a correct result". In particular, consider:

```
type ARR is array (INTEGER range <>) of CHARACTER;
NS : constant ARR := "";
```

Can the upper bound of the null string literal be computed as INTEGER'FIRST + length("") - 1, which will cause NUMERIC_ERROR to be raised?

!recommendation 85-03-11

NUMERIC_ERROR must not be raised when computing the upper bound of a null string literal, but see AI-00325.

!discussion 85-03-11

4.2(3) says explicitly that the upper bound of a null string literal is "the predecessor, as given by the PRED attribute, of the lower bound. ... CONSTRAINT_ERROR is raised if the lower bound does not have a predecessor (see 3.5.5)". Moreover, 3.5.5(9) says that CONSTRAINT_ERROR is raised by PRED if its argument equals 'FIRST of the base type, i.e., INTEGER'PRED(INTEGER'FIRST) is required to raise CONSTRAINT_ERROR, regardless of what predefined operations are actually used to implement the 'PRED attribute.

As usual, when a specific rule exists, it is used in place of a general rule. In this case, the specific rules given in 4.2(3) and 3.5.5(9) should not be ignored in favor of the general rule given in 11.1(6). Unless there are strong reasons why an implementation cannot avoid raising NUMERIC_ERROR, NUMERIC_ERROR should not be raised (see AI-00325).

Ada Commentary ai-00312-ra.wj downloaded on Wed May 11 15:45:03 EDT 1988

!standard 11.01 (06) 86-07-23 AI-00312/04
!class ramification 84-11-26
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-26
!status work-item 84-11-01
!status received 84-10-31
!references 83-00463
!topic NUMERIC_ERROR when evaluating null aggregates and slices

!summary 85-01-04

When determining the length of a null aggregate or slice, it is usually easy for an implementation to avoid raising NUMERIC_ERROR. This exception should be raised in these circumstances only when relevant restrictions in the execution or compilation environment make it impractical or impossible to avoid raising the exception (see AI-00325).

!question 85-01-04

11.1(6) says that NUMERIC_ERROR is raised when an implementation "uses a predefined numeric operation for the execution, evaluation, or elaboration of some construct," and the operation "cannot deliver a correct result." Some constructs can be implemented in several ways, some of which raise NUMERIC_ERROR. In particular, consider:

```
type ARR is array (INTEGER range <>) of INTEGER;
VAR1 : ARR (1..0);
...
VAR1 := (5..INTEGER'FIRST => 0);
```

Can NUMERIC_ERROR be raised by this assignment? (An implementation could perform the required length check by first evaluating upper_bound - lower_bound + 1, which would raise NUMERIC_ERROR in this case.)

Suppose the range appeared in a slice, e.g.,

```
VAR2 : ARR (5..10);
...
VAR1 := VAR2 (5..INTEGER'FIRST);
```

Can NUMERIC_ERROR be raised when the slice's length is determined?

!response 85-01-04

No exception would be raised if an implementation first checked to see if an upper bound is less than the lower bound. For example, the length of an aggregate could be computed as follows:

```
if ub < lb then
     length := 0;
else
     length := ub - lb + 1;
end if;
```

NUMERIC_ERROR would be raised only if the length were computed before checking to see if the computed length is zero or negative.

AI-00325 discusses general principles under which implementations should be allowed to take advantage of rules such as those in 11.1(6). In the examples considered here, it appears unlikely that a convincing justification (in terms of AI-00325) could be made for raising NUMERIC_ERROR, and consequently, any compiler raising NUMERIC_ERROR in these situations should be considered unacceptable. However, as AI-00325 points out, the judgment of acceptability for a particular implementation should be based on relevant restrictions in the execution or compilation environment that make it impossible or impractical to avoid raising the exception.

Ada Commentary ai-00313-bi.wj downloaded on Wed May 11 15:50:02 EDT 1988

!standard 04.03.02 (11) 86-07-23 AI-00313/03
!standard 04.06 (13)
!class binding interpretation 84-11-05
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-26
!status work-item 84-11-05
!status received 84-11-05
!references 83-00466, 83-00467, 83-00468
!topic Non-null bounds belong to the index subtype

!summary 84-12-28

CONSTRAINT_ERROR is raised if a non-null choice of an aggregate does not belong to the corresponding index subtype.

For conversion to an unconstrained array type, CONSTRAINT_ERROR is raised if a non-null dimension of the operand has bounds that do not belong to the corresponding index subtype of the target type.

!question 84-12-28

Consider the following example:

```
subtype INT is INTEGER range 1..8;
subtype SM_INT is INTEGER range 1..7;
type TWO_DIM is array (INT range <>, INT range <>) of INTEGER;
type SMALL is array (SM_INT range <>, INT range <>) of INTEGER;
procedure P (X : TWO_DIM);
...
P ( (1..9 => (1..0 => 3)) );                    -- CONSTRAINT_ERROR?
... SMALL (TWO_DIM'(7..8 => (1..0 => 0)) ) ... -- CONSTRAINT_ERROR?
```

Should CONSTRAINT_ERROR be raised by the aggregate in the call, since 9 does not belong to the index subtype, INT? Similarly, should CONSTRAINT_ERROR be raised by the conversion to SMALL since 8 does not belong to the index subtype, SM_INT?

!recommendation 84-12-28

For the evaluation of a non-null one-dimensional aggregate or a non-null dimension of a multidimensional aggregate, a check is made that the index values defined by choices belong to the corresponding index subtype. The exception CONSTRAINT_ERROR is raised if this check fails.

For the evaluation of a conversion to an unconstrained array type, a check is made that the bounds of each non-null dimension of the operand belong to the corresponding index subtype of the target type. CONSTRAINT_ERROR is raised if this check fails.

!discussion 84-12-28

4.3.2(11) says:

> For the evaluation of an aggregate that is not a null array, a check is made that the index values defined by choices belong to the corresponding index subtypes ...

4.3.2(2) says:

> In what follows, the rules concerning array aggregates are formulated in terms of one-dimensional aggregates.

4.3.2(2) can be interpreted to mean that whenever later rules use the unqualified term, "aggregate," the rule applies equally to one-dimensional and to multidimensional aggregates. Alternatively, the rule can be understood to mean that unless explicitly stated otherwise, rules using the unqualified term "aggregate" refer both to aggregates having a one-dimensional array type and to subaggregates considered as one-dimensional aggregates. In general, the wording of 4.3.2 supports the second interpretation, i.e., that the unqualified term "aggregate" means either an aggregate having a one-dimensional array type or a dimension of a multidimensional aggregate.

If the rule in 4.3.2(11) is therefore applied to each dimension of a multidimensional aggregate, then CONSTRAINT_ERROR should be raised by the aggregate in the call to P (since the outermost subaggregate is non-null, considered solely as a one-dimensional aggregate). On the other hand, use of the term "index subtypes" in 4.3.2(11) suggests that the rule applies to null multidimensional aggregates as a whole; with this interpretation, no exception would be raised by the example.

Conversions provide another way of creating non-null index bounds that do not belong to an index subtype. When converting to an unconstrained array type, 4.6(13) says:

> "If the type mark denotes an unconstrained array type and if the operand is not a null array, then for each index position, a check is made that the bounds of the result belong to the corresponding index subtype of the target type."

In other words, if the operand is null, no check is made that non-null bounds belong to an index subtype. In the example given in the question, the operand of the conversion to SMALL is a null two-dimensional array. Hence, no check is made to see if 7..8 belongs to SM_INT, and so no exception is raised by the conversion.

Passing null array aggregates as subprogram or entry parameters and converting null array values to an unconstrained array type are the only ways of creating non-null index bounds that lie outside an index subtype. In particular, note that when an index constraint for a multi-dimensional array type is elaborated, each discrete range is checked for compatibility with the

Ada Commentary ai-00319-co.wj downloaded on Wed May 11 16:45:03 EDT 1988

corresponding index subtype, regardless of whether any of the discrete ranges are null [3.6.1(4)]. Moreover, when converting to a constrained array type, the bounds of the target type are used for the result of the conversion [4.6(11)], so the bounds necessarily belong to each index subtype.

On the whole, the intent of the language is to ensure that values belong to a subtype. In particular, a non-null index range should always be compatible with its index subtype (i.e., each bound should belong to its index subtype). The wording in 4.3.2(11) is ambiguous, but should be interpreted to achieve the intent of the design. The wording in 4.6(13) is incorrect and does not reflect the intent of the design; the wording should be understood to require that non-null index bounds always belong to their index subtype.

Tests in Version 1.4 and 1.5 of the validation suite require that CONSTRAINT_ERROR not be raised for null multidimensional array aggregates, thus allowing the bounds of non-null index ranges to lie outside the the corresponding index subtype. These tests should be corrected.

Ada Commentary ai-00314-co.wj downloaded on Wed May 11 15:50:03 EDT 1988

!standard 03.05.07 (09) 87-08-06 AI-00314/05
!class confirmation 84-11-24
!status approved by WG9/AJPO 87-07-30
!status approved by Director, AJPO 87-07-30
!status approved by Ada Board 87-07-30
!status approved by WG9 87-05-29
!status panel/committee-approved 87-01-19 (reviewed)
!status panel/committee-approved (6-0-0) 86-11-14 (pending editorial review)
!status work-item 86-10-13
!status received 84-11-24
!references 83-00473
!topic The safe numbers for IBM-370 floating point

!summary 86-10-13

The safe and model numbers for IBM-370 32-bit floating point have the following characteristics:

```
DIGITS    =   6
MANTISSA  =  21
EMAX      =  84
SAFE_EMAX = 252
```

!question 86-10-13

What are the safe and model numbers for a machine like the IBM-370 whose floating point representation is hexadecimal (24 bits, plus sign) with exponent values ranging from -64 to 63. What are the values of DIGITS, MANTISSA, EMAX, and SAFE_EMAX for such a machine?

!response 86-10-13

The representation of a 24-bit hexadecimal mantissa is .HHHHHH. Since the first three bits of the mantissa can be zero, the minimum number of significant binary digits is 21. According to the formula given in 3.5.7(6), 21 mantissa bits supports 6 decimal digits of accuracy, since log(10)/log(2) = 3.32, and the integer next above 6*3.32 + 1 is 21.

Given that MANTISSA is 21, the exponent range must be at least 4*21, for a binary mantissa. The smallest positive floating point number is 16#0.100000#E-64, which is equivalent to 2#0.1#E-(256+3). The largest positive floating point number is 16#0.FFFFFF#E+63, which is equivalent to 2#0.1...1#E+(252). Since the exponent range for the safe numbers must be symmetric, the exponent for the largest safe number is min(252, 256+3) = 252. Hence, the following parameters describe the model and safe numbers for the 32-bit IBM-370 floating point representation:

```
DIGITS    =   6
MANTISSA  =  21
EMAX      =  84
SAFE_EMAX = 252
```

Ada Commentary ai-00316-ra.wj downloaded on Wed May 11 16:45:02 EDT 1988

!standard 14.03.09 (06) 87-06-18 AI-00316/05
!class ramification 84-11-24
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (5-0-0) 86-09-11 (pending editorial review)
!status work-item 86-08-12
!status received 84-11-24
!references 83-00470
!topic Definition of blank, inclusion of horizontal tab

!summary 86-08-12

A blank is defined as a space or a horizontal tabulation character.

!question 86-09-17

The GET procedures for INTEGER_IO (14.3.7(6)), for REAL types (14.3.8(9)), and for ENUMERATION_IO (14.3.9(6)) skip leading blanks, line terminators, or page terminators.

14.3.5 contains general comments about GET and PUT procedures. In particular, 14.3.5(5) specifies for ENUMERATION_IO that:

> a blank [is] defined as a space or horizontal tabulation character.

Does 14.3.5(5) apply only to ENUMERATION_IO, or was it the intention that horizontal tabs be skipped in INTEGER_IO, FLOAT_IO, and FIXED_IO as well?

!response 86-08-17

14.3.5(5) states:

> Any GET procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators; a blank being defined as a space or a horizontal tabulation character.

It would seem that the definition of blank is specific to ENUMERATION_IO, but 14.3.5(6) states:

> For a numeric type, the GET procedures have a format parameter called WIDTH. If the value given for this parameter is zero, the GET procedure proceeds in the same manner as for enumeration types, ...; any skipped leading blanks are included in the count.

This indicates that the definition is not specific to ENUMERATION_IO. Thus, for TEXT_IO, a blank is defined as a space or a horizontal tabulation character.

!standard 03.07.02 (05) 87-06-18 AI-00319/09
!standard 03.08.01 (04)
!class confirmation 84-06-28
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (24-0-1) 87-02-19
!status panel/committee-approved 86-10-15 (reviewed)
!status panel/committee-approved (6-1-1) 86-09-10 (pending editorial review)
!status work-item 86-07-24
!status returned to committee by WG9 86-05-09
!status committee-approved (7-1-1) 85-05-18
!status work-item 85-01-26
!status committee-approved 84-06-28
!references AI-00007
!topic Checking for subtype incompatibility

!summary 86-09-13

No object can have a subcomponent with an incompatible discriminant or index constraint. In particular, even when a discriminant constraint is applied to a private type before its full declaration or to an incomplete type (before its full declaration) and a discriminant is used to constrain a subcomponent, no object of the type can be created if it would have a subcomponent with an incompatible discriminant or index constraint.

!question 85-01-26

When a discriminant constraint is applied to a private type before its full declaration or to an incomplete type before its full declaration, 3.3.2(8) and 3.7.2(5) require that the value of any discriminant used to constrain a subcomponent be checked for compatibility with the subcomponent's type. Since no subcomponents exist prior to the full declaration, can the required check be omitted in this case?

!response 87-06-04

A general principle underlying Ada's design is that if an object has a defined value, it must belong to the object's subtype. In particular, since a discriminant always has a defined value, its value must always belong to the discriminant's subtype, even when a discriminant is a subcomponent of a composite type. Similarly, non-null index bounds must belong to the index subtype. This implies that no object is allowed to have a subcomponent with a discriminant or index bound that is incompatible with its type.

Ada Commentary ai-00320-bi.wj downloaded on Wed May 11 16:50:01 EDT 1988

| !standard 14.03.05 (03) 86-07-23 AI-00320/06
!standard 14.02.04 (00)
!standard 14.03.04 (00)
!standard 14.02.02 (00)
!class binding interpretation 84-01-17
| !status approved by WG9/AJPO 86-07-22
| !status approved by Director, AJPO 86-07-22
| !status approved by WG9/Ada Board 86-07-22
| !status approved by Ada Board 86-07-22
| !status approved by WG9/ADA Board 85-05-13
!status committee-approved (9-2-1) 85-05 (by letter ballot)
!status committee-approved (5-1-2) 85-02-25 (pending letter ballot)
!status work-item 84-11-06
!status received 84-01-17
!references 83-00465, 83-00501
!topic Sharing external files

!summary 85-04-01

If several file objects are associated with the same external file, some effects are implementation dependent. For example, if two sequential file objects are associated with the same external file, applying a read or write operation to one file object can change the effect of applying these operations (or the end of file operation) to the other file object. Other effects are specified by the language. In particular, if two text file objects are associated with a single external file (e.g., a terminal), the page, line, and column numbers for the output file object cannot be updated implicitly after reading from the input file object, and vice versa.

!question 85-03-11

If an external file is associated with different file objects, can operations on one file object affect operations on the other object? In particular, if two text file objects are associated with the same external file (e.g., a terminal), can the page, line, and column numbers for one of the file objects be updated implicitly as a result of operations on the other file object?

!recommendation 85-04-01

If the same external file is associated with more than one file object, the following effects are specified by the language (other effects are implementation dependent):

> Operations on one text file object do not affect the column, line, and page numbers of any other file object.
>
> STANDARD_INPUT and STANDARD_OUTPUT are associated with distinct external files, so operations on one of these files cannot affect operations on the other file. In particular, reading from STANDARD_INPUT does not affect the current page, line, and column numbers for STANDARD_OUTPUT, nor does writing to STANDARD_OUTPUT affect the current page, line, and column

numbers for STANDARD_INPUT.

For direct files, the current index is a property of each file object; an operation on one direct file object does not affect the current index of any other direct file object.

For direct files, the current size of the file is a property of the external file.

!discussion 85-03-11

The Standard notes in 14.1(13) that the effect of sharing an external sequential file by several file objects is implementation dependent if one file object has mode IN_FILE and the other, mode OUT_FILE. Although this statement is made in a Note, it indicates the intent of the Standard, which is to leave undefined most effects of sharing an external file among several file objects.

For example, since the effect of sharing an external file is not fully defined by the Standard, if an external file is associated with two sequential file objects having mode IN_FILE, a read operation using one of the file objects could cause END_OF_FILE to become true for the other object. Similarly, the data read for one of the file objects could be affected by how many read operations have been applied to the other object.

Some effects of sharing external files, however, are specified explicitly. In particular, 14.2(4) says, "The current index of a direct file is a property of a file object, not of an external file." This wording is intended to mean that if the same external file is associated with two direct file objects, each file object maintains its own current index for purposes of reading and writing elements of the external file. A read operation applied to one file object will not affect the current index of the other file object.

The Standard intends the phrase "P is a property of Y and not a property of Z" to mean "P is potentially affected by operations on Y and not by operations on Z." Technically, the phrase need not be interpreted this way (for example, height is a property of people and not of the food they eat, but diet can certainly have an effect on height). However, the intent of the Standard is clear.

Some effects are specified to apply to an external file regardless of whether the file is shared. In particular, 14.2(3) says the current size of a direct file "is a property of the external file." If a write operation increases the size of an external file, the result returned by the SIZE function will reflect the current size of an external file, regardless of what file object is used to access the external file.

For text files, line, page, and column numbers are properties affected by read and write operations. These are properties of file objects: 14.1(2) says, "In the remainder of this Chapter, the term FILE is always used to refer to a file object; the term EXTERNAL FILE is used otherwise." 14.3.5(3)

says, "All procedures GET and PUT maintain the current column, line, and page numbers of the specified file: ..." Since the 14.3.5(3) sentence uses the term "file" instead of the term "external file," current numbers are not properties of the external file. The intended interpretation, therefore, is that operations using one text file object must not change the current numbers of any other text file object, even if both objects are associated with the same external file.

This position is consistent with the treatment of STANDARD_INPUT and STANDARD_OUTPUT. 14.3(5) says these files "are associated with two implementation-defined external files." Since these files are independent external files, operations on STANDARD_INPUT could not affect the current numbers for STANDARD_OUTPUT (and vice versa) even if these numbers were properties of the external files.

It might be argued that it would be more convenient for Ada programmers if the current numbers were associated with the external file. Even if this were the case, the current numbers would not always reflect the actual position of the cursor for a terminal. For example,

```
OPEN (Input_Text_File, In_File, "Terminal");
-- read integer from terminal, e.g., the integer 1000
GET (Input_Text_File, Integer_Item);
```

At this point, if the integer given at the terminal is terminated by an end-of-line, the cursor is at column 1 of line 2, but according to Ada semantics, since the line terminator has not yet been read, the function LINE must return the value 1 and COL must return 5. This example merely shows that when TEXT_IO is used with terminals, the current numbers are not as useful as they might at first appear. Associating these numbers with external files rather than with file objects would not solve as many problems as one might think.

Ada Commentary ai-00321-bi.wj downloaded on Wed May 11 16:50:03 EDT 1988

!standard 13.01 (06) 86-07-23 AI-00321/02
!class binding interpretation 83-10-30
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-11-05
!status received 84-01-29
!references 83-00449
!topic Forcing occurrence of index subtype

!summary 85-01-02

A forcing occurrence of the name of an array type or subtype forces the default determination of each index subtype, and similarly, for forcing occurrences of any type or subtype having a subcomponent of such an array type.

!question 85-01-02

13.1(6) attempts to prevent uses of an entity before its representation is fully determined by a representation clause or by the full declaration of a private type. The current rules do not seem to cover occurrences of a type as an index subtype:

```
type T is (A, B, C);
type ARR is array (T range <>) of INTEGER;
OBJ : ARR (A..C);
for T use (1, 2, 5);          -- legal?
```

ARR does not have any subcomponents of type T; is the use of ARR to declare OBJ nonetheless a forcing occurrence for T?

!recommendation 85-01-02

A forcing occurrence of the name of an array type or subtype forces the default determination of each index subtype, and similarly, for forcing occurrences of any type or subtype having a subcomponent of such an array type.

!discussion 85-01-02

A review of the motivation for the rule in 13.1(6) shows that forcing occurrences of the name of an array type should be considered forcing occurrences for each index subtype, and similarly for forcing occurrences of the name of a type with subcomponents having such an array type. The current rule is inadequate, since it does not mention index subtypes.

Ada Commentary ai-00322-bi.wj downloaded on Wed May 11 17:45:01 EDT 1988

| !standard 13.01 (06) 86-07-23 AI-00322/02
| !standard 02.08 (09)
| !class binding interpretation 83-10-30
| !status approved by WG9/AJPO 86-07-22
| !status approved by Director, AJPO 86-07-22
| !status approved by WG9/Ada Board 86-07-22
| !status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-02-26
!status committee-approved 84-11-28
!status work-item 84-11-05
!status received 84-01-29
| !references AI-00186, 83-00450
!topic Forcing occurrences in unknown pragmas

!summary 84-12-26

An occurrence of a name within an expression is not a forcing occurrence if the expression occurs in a pragma whose identifier is not defined either by the Standard or by the implementation.

!question 85-01-26

13.1(6) attempts to prevent uses of an entity before its representation is fully determined by a representation clause. However, consider the occurrence of a name in an implementation-defined pragma:

```
type YES is new INTEGER range 1..10;
pragma DEBUG (YES);           -- forcing occurrence?
pragma DEBUG (10 in YES);     -- forcing occurrence?
for YES'SIZE use 8;
```

In the first pragma, YES cannot be interpreted as a primary in an expression, so this is not a forcing occurrence. In the second pragma, YES appears in an expression, so even if the pragma is undefined and "has no effect" [2.8(9)], the representation clause seems to be illegal. Is this analysis correct?

!recommendation 84-12-26

An occurrence of a name within an expression of a pragma is not a forcing occurrence if the pragma's identifier is not defined for the implementation.

!discussion 85-01-26

2.8(9) says,

> "A pragma that is not language-defined has no effect if its identifier is not recognized by the (current) implementation.

13.1(6) says,

> "A forcing occurrence is any occurrence other than [various places including within a pragma]. In any case, an occurrence

within an expression is always forcing."

The final phrase implies that occurrence of a name in the expression of a pragma is a forcing occurrence.

If a name occurs in an expression of a pragma that has no effect according to 2.8(9), can such an occurrence be considered a forcing occurrence according to 13.1(6), i.e., can the pragma have an effect on the legality of a later representation clause?

The intention of 2.8(9) was to ensure that implementation-defined pragmas do not have any effect except for those implementations that recognize them. In particular, the presence of an unrecognized implementation-defined pragma should not make a compilation unit illegal. It is inconsistent with this intent to have an unrecognized pragma affect the legality of a representation clause. Therefore, if a pragma is not defined for an implementation, occurrence of a name within an expression of the pragma should not be considered a forcing occurrence.

Ada Commentary ai-00325-ra.wj downloaded on Wed May 11 17:45:02 EDT 1988

!standard 01.01.02 (00) 86-07-23 AI-00325/04
!class ramification 84-11-28
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9/ADA Board 85-05-13
!status committee-approved (9-2-1) 85-02 (by letter ballot)
!topic Implementation-dependent limitations

!summary 85-01-26

Implementation-dependent limitations must be justified. An implementation-dependent limitation is justified if it is impossible or impractical to remove it, given an implementation's execution environment.

!question 85-05-27

Under what conditions can an implementation fail to support some construct specified in the Standard, either by rejecting the program at compile time or by raising an exception?

!response 85-05-27

The goal of providing a common programming language requires that differences between implementations be minimized, and yet a strict reading of the Standard allows implementation dependences in the processing of many constructs, leading to potentially surprising and undesirable differences between implementations. In particular, there are many ways an implementation can refuse to support some construct either by refusing to accept the construct at compile time or by raising an exception at run-time. The following is a partial list of such possibilities:

raising NUMERIC_ERROR as permitted by 11.1(6)
raising USE_ERROR for file operations
raising STORAGE_ERROR at any point
limiting SYSTEM.MAX_INT (e.g., to 6)
limiting line size to an unreasonably small value
limiting the number of tasks to one (the main program)
limiting the number of lines in a subprogram
etc.

From a technical point of view, such considerations could be used to reject almost any Ada program while maintaining conformance to the Standard. The Standard is permissive in these respects because specification of acceptable limits was felt to be undesirable, even if possible, in the definition of the language.

On the other hand, acceptance of an implementation depends on it behaving in a reasonable manner. In a sense every test in the validation suite is a capacity test that implementations are expected to pass. If any implementation fails one of these tests on grounds of capacity limitations or

by raising unexpected exceptions such as NUMERIC_ERROR, the failure must be justified on the grounds that successful passing of the test is impossible or impractical in the test environment.

For example, an implementation with no I/O devices cannot be expected to implement I/O. On the other hand, an implementation running on a machine with conventional disks should not be permitted to "escape" the I/O tests by raising USE_ERROR. Similarly, all the implementations validated so far lack garbage collectors and thus raise STORAGE_ERROR where no exception would normally be expected. Such a limitation can be considered acceptable if it is argued that including a garbage collector is impractical; moreover, the Standard allows such a limitation [4.8(7)]. But an implementation that raised STORAGE_ERROR on every procedure call would not be considered acceptable.

It is within the domain of the Language Maintenance Committee to make recommendations about what should be considered acceptable implementation dependent interpretations of the Standard. The Ada Validation Office or agencies procuring implementations may wish to take these recommendations into consideration when deciding whether to approve or accept a particular implementation.

Ada Commentary ai-00328-ra.wj downloaded on Wed May 11 17:50:01 EDT 1988

!standard 12.02 (01) 87-03-16 AI-00328/08
!class ramification 85-04-15
!status approved by WG9/AJPO 87-03-10
!status approved by Director, AJPO 87-03-10
!status approved by Ada Board (25-0-0) 87-02-19
!status approved by WG9 85-11-18
!status committee-approved (as corrected) 85-09-04
!status committee-approved (11-0-0) 85-05-17 (subject to editorial revision)
!status work-item 85-04-15
!status received 85-02-15
!references 83-00502, 83-00503, 83-00505, 83-00599, 83-00612
!topic Legality of uninstantiated generic units

!summary 87-02-23

The legality of a generic unit must be checked even if the generic unit is never instantiated.

!question 87-02-23

Consider the following example:

```
generic
     type PRIV (D : INTEGER) is private;
procedure OUTER;

procedure OUTER is
     generic
          type PV is private;
     procedure INNER;

     procedure INNER is
          OBJ : PV;
     begin
          ...
     end INNER;

     procedure NEW_INNER is new INNER(PRIV); -- illegal?
begin
     ...
end OUTER;
```

Is the indicated instantiation illegal because the object declaration inside INNER is illegal when the actual generic parameter is an unconstrained type with discriminants that do not have defaults [12.3.2(4)], or is it unnecessary to detect the error until the OUTER unit is instantiated? Since a generic unit is only a template for non-generic units, it should be possible to defer certain legality checks until an instantiation is attempted, i.e., the instantiation above should not cause the declaration of OUTER to be rejected.

!response 85-04-15

The legality of an instantiation as expressed in 12.3.2(4) is not dependent on whether the instantiation occurs inside a generic unit. In the example, since the declaration of PRIV shows that PRIV is an unconstrained type with discriminants that do not have defaults, the rule in 12.3.2(4) applies, and the instantiation is illegal. In particular, if OUTER is a library unit, it must be rejected and not added to the program library [10.3(3)].

Ada Commentary ai-00330-bi.wj downloaded on Wed May 11 17:50:02 EDT 1988

!standard 08.03 (17) 86-07-23 AI-00330/12
!standard 03.05.01 (03)
!standard 03.03.03 (01)
!class binding interpretation 85-03-10
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 86-05-09
!status committee-approved 86-05 (8-0-3) (by ballot)
!status committee-approved (7-0-2) 86-02-21 (pending letter ballot)
!status reconsidered by committee 85-02-20 (5-1-2)
!status committee-approved (7-0-3) 85-11-20 (subject to editorial review)
!status failed letter ballot (5-5-2)
!status committee-approved 85-09-05 (4-2-3) (pending letter ballot)
!status work-item 85-03-10
!status received 85-03-10
!references AI-00002, 83-00511, 83-00512, 83-00513, 83-00514, 83-00515, 83-00516, 83-00560, 83-00590, 83-00580, 83-00597, 83-00629, 83-00630, 83-00633, 83-00648, 83-00656, 83-00655, 83-00667, 83-00701, 83-00709, 83-00706
!topic Explicit declaration of enumeration literals

!summary 86-03-10

If an enumeration literal is declared with an enumeration type definition, then a function having the same identifier as the enumeration literal and the same parameter and result type profile cannot also be declared immediately within the same declarative region. Similarly, a non-overloadable declaration of the enumeration literal's identifier is not allowed immediately within the declarative region containing the enumeration type definition.

!question 86-03-10

Consider:

```
type ENUM is (P, Q);
function P return ENUM;          -- legal? (no)
type Q is range 1..10;           -- legal? (no)
```

Are the second declarations of P or Q legal?

!recommendation 86-03-10

If an enumeration literal is declared with an enumeration type definition, then a homograph of the enumeration literal must not be declared immediately within the same declarative region.

!discussion 86-03-10

8.3(17) allows the declaration of homographs immediately within the same

declarative region if "exactly one of them is the implicit declaration of a predefined operation" or "exactly one of them is the implicit declaration of a derived subprogram." Is an enumeration literal that appears (explicitly) in an enumeration literal specification:

- a predefined operation?
- implicitly declared?

A careful reading of the Standard suggests that enumeration literals declared in an enumeration literal specification are implicitly declared operations. Section 3.3.3 distinguishes between explicitly and implicitly declared operations. Enumeration literals are mentioned only in the paragraph that defines the implicitly declared operations [3.3.3(2)]. If the intent had been that enumeration literals be considered explicitly declared operations when they are declared in an enumeration literal specification, this presumably would have been stated in the only paragraph that discusses explicitly declared operations [3.3.3(1)].

3.5.1(3) says that an enumeration literal specification "is equivalent to the declaration of a parameterless function." This wording leaves open whether the declaration is to be considered explicit or not.

Is an enumeration literal a predefined operation? One could consider an enumeration literal to be a predefined operation on the grounds that once an enumeration literal specification is written, the effect of each enumeration literal is completely defined by the rules of the Standard. A similar effect happens for component selection operations. The effect (and existence) of the operation for selecting, say, a component named A depends on what the programmer writes for a type definition; once the type definition is written, the effect is determined by the rules of the Standard. Similarly, it is reasonable to consider enumeration literals to be predefined operations.

However, if we consider enumeration literals to be implicitly declared predefined operations, then explicit declarations like those of function P and type Q (in the question) are allowed by 8.3(17). It was not the intent to allow such declarations. Comment #2817 explicitly asked whether declarations like the declarations of P were allowed by the Standard. The response was that the two declarations of P were not allowed within the same declarative region, i.e., the intent was to prohibit such declarations. If such declarations were not to be allowed, clearly the declarations of Q were also not to be allowed. Such declarations are likely to be programmer errors and were intended to be forbidden.

Ada Commentary ai-00331-ra.wj downloaded on Wed May 11 18:45:02 EDT 1988

!standard 04.08 (05) 87-06-18 AI-00331/07
!class ramification 85-04-24
!status approved by WG9/AJPO 87-06-17
!status approved by Director, AJPO 87-06-17
!status approved by WG9 87-05-29
!status approved by Ada Board (21-0-0) 87-02-19
!status panel/committee-approved 86-11-14 (reviewed)
!status panel/committee-approved (8-0-1) 86-09-10 (pending editorial review)
!status work-item 86-04-11
!status received 85-04-24
!references AI-00397, AI-00324, 83-00527, 83-00536
!topic The effect of a constraint in an allocator

!summary 86-09-28

When a discriminant or index constraint is imposed on the type mark in an allocator and the type mark denotes an access type, the constraint does not affect the subtype of the allocated object (which in this case has an access value).

Similarly, when the type mark in an allocator denotes a scalar type, the subtype denoted by the type mark does not affect the subtype of the allocated (scalar) object.

!question 86-09-28

Does a constraint given in an allocator apply to the designated object when the designated type is an access or a scalar type? Consider the following example:

```
declare
     type ACC_STR is access STRING;
     type ACC_ACC_STR is access ACC_STR;
     V1 : ACC_ACC_STR;
begin
     V1 := new ACC_STR (1..2);          -- (1)
     V1.all := new STRING'("123");      -- CONSTRAINT_ERROR? (no)
end;
```

Does the constraint given in the allocator at (1) apply to the designated object, V1.all? If so, the assignment to V1.all should raise CONSTRAINT_ERROR.

A similar situation can arise for scalar designated types:

```
subtype S10 is INTEGER range 1..10;
subtype S11 is INTEGER range 1..11;
type ACC_S11 is access S11;
V2 : ACC_S11;
...
V2 := new S10;
```

```
V2.all := 11;          -- CONSTRAINT_ERROR? (no)
```

Does the constraint given in the allocator, new S10, apply to the designated object, V2.all? If so, CONSTRAINT_ERROR should be raised by the assignment to V2.all.

!response 86-09-28

RM 4.8(5) states:

> If the type of the created object is an array type or a type with discriminants, then the created object is always constrained. If the allocator includes a subtype indication, the created object is constrained either by the subtype or by the default discriminant values. If the allocator includes a qualified expression, the created object is constrained by the bounds or discriminants of the initial value. For other types, the subtype of the created object is the subtype defined by the subtype indication of the access type definition.

The first three sentences apply only to array objects and objects with discriminants. These sentences specify the constraint on a created object when its type is an array type or a type with discriminants.

The last sentence applies to scalar, access, and private or record types without discriminants. For such designated types, the allocator imposes no constraints on the designated object: the subtype of the created object is that specified in the definition of the access type. This means that in the example:

```
V1 := new ACC_STR(1..2);
```

the subtype of the object created by the allocator is ACC_STR, since the designated object has an access type, and ACC_STR is the subtype specified in ACC_ACC_STR's type declaration. (Of course, even though the index constraint is not imposed on the designated object, the subtype indication, ACC_STR(1..2), must be evaluated and checked. In this case, since 1..2 is compatible with ACC_STR's designated type, no exception is raised.) The subsequent assignment to V1.all only checks that the assigned access value belongs to the type ACC_STR, and no exception is raised.

Similarly, the allocator, new S10, imposes no additional constraint on the allocated object, so the assignment of 11 to V2.all does not raise an exception. An attempt to assign 12 to V2.all would, however, raise CONSTRAINT_ERROR because the constraint specified in ACC_S11's subtype indication only allows values in the range 1..11.

Ada Commentary ai-00332-bi.wj downloaded on Wed May 11 18:45:03 EDT 1988

!standard 14.02.01 (04) 86-07-23 AI-00332/04
!standard 14.02.01 (07)
!standard 14.04 (04)
!standard 14.04 (05)
!class binding interpretation 85-04-24
!status approved by WG9/AJPO 86-07-22
!status approved by Director, AJPO 86-07-22
!status approved by WG9/Ada Board 86-07-22
!status approved by Ada Board 86-07-22
!status approved by WG9 85-11-18
!status committee-approved (7-1-0) 85-05-18
!status work-item 85-04-24
!status received 85-04-24
!references 83-00522, 83-00523, 83-00524, 83-00525
!topic NAME_ERROR or USE_ERROR raised when I/O not supported

!summary 85-07-21

CREATE and OPEN can raise USE_ERROR or NAME_ERROR if file creation or opening is not allowed for any file.

!question 85-07-21

Suppose an implementation does not support the creation or use of external files (because there are no devices for storing such files). What exception should be raised by a call to CREATE or OPEN? NAME_ERROR might be raised on the grounds that the name string does not allow the identification of an external file; USE_ERROR might be raised because files with sequential, direct, or text file characteristics cannot be created or exist. Which is the appropriate exception to raise?

!recommendation 85-07-21

The Standard does not specify whether a call to CREATE or OPEN will raise NAME_ERROR or USE_ERROR if an implementation does not allow any files to be created or opened.

!discussion 85-07-21

14.2.1(7) says (for OPEN):

> The exception NAME_ERROR is raised if the string given as NAME does not allow the identification of an external file; in particular, this exception is raised if no external file with the given name exists. The exception USE_ERROR is raised if, for the specified mode, the environment does not support creation of an external file with the given name (in the absence of NAME_ERROR) and form.

Similar wording apears in 14.2.1(4) for CREATE. In addition, 14.4(1) says:

> If more than one error condition exists, the corresponding

> exception that appears earliest in the following list is the one that is raised.

Then 14.4(4) says:

> The exception NAME_ERROR is raised by a call of CREATE or OPEN if the string given for the parameter NAME does not allow the identification of an external file. For example, this exception is raised if the string is improper, or, alternatively, if either none or more than one external file corresponds to the string.

14.4(5) (which appears after the definition of NAME_ERROR) says:

> The exception USE_ERROR is raised if an operation is attempted that is not possible for reasons that depend on characteristics of the external file. For example, this exception is raised by the procedure CREATE ... if the parameter FORM specifies invalid access rights ... .

If no external file can be created or opened, then the Standard clearly allows NAME_ERROR to be raised in preference to USE_ERROR on the grounds that no external file can be identified with the specified name.

On the other hand, an implementation could take the position that all names are legal but no files can be created or opened because of characteristics other than the name being used. This position implies that calls to OPEN and CREATE raise USE_ERROR in preference to NAME_ERROR.

Either of these positions is consistent with the requirements of the Standard. The main purpose of this Commentary is to point out that USE_ERROR can be raised by OPEN and CREATE when no file I/O is supported.

(In discussing this issue, it was noted that when no files can be created or opened, every attempt to create a temporary file will raise USE_ERROR; NAME_ERROR cannot be raised because no name is specified.)

Ada Commentary ai-00339-bi.wj downloaded on Wed May 11 18:50:02 EDT 1988

!standard 02.07 (01) 86-12-01 AI-00339/04
!standard 02.01 (01)
!class binding interpretation 86-02-20
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (7-0-1) 86-02-20 (pending editorial review)
!status received 85-06-18
!references 83-00557
!topic Allow non-English characters in comments

!summary 86-06-19

An implementation is allowed (but not required) to accept an extended character set (i.e., graphic characters whose codes do not belong to the ISO seven-bit coded character set (ISO standard 646)) as long as the additional characters appear only in comments.

!question 86-05-05

Are the graphic characters in comments limited to those in the ISO seven-bit coded character set? Note that additional characters are needed to write comments in languages other than English (e.g., French or Japanese), and that since comments have no influence on the legality of a program [2.7(1)], additional graphic characters seem to be allowed.

!recommendation 86-06-19

An implementation is allowed to accept an extended character set (i.e., graphic characters whose codes do not belong to the ISO seven-bit coded character set (ISO standard 646)) as long as the additional characters appear only in comments (i.e., the additional characters only appear after two adjacent hyphens and precede the end of the line).

!discussion 86-06-19

2.1(1) says:

> The only characters allowed in the text of a program are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the ISO seven-bit coded character set (ISO standard 646), and is represented (visually) by a graphical symbol.

This wording suggests that a program is illegal if it contains any character that is not either a graphic character or a format effector. For example, an occurrence of the character code for ASCII.NUL in a compilation would presumably not be allowed. Of course, an actual program might be coded using eight or nine-bit codes as long as the only allowed codes are those corresponding to the graphic characters and format effectors of the ISO standard.

Such an interpretation of 2.1(1) is unduly restrictive for comments. When writing a comment, one should be able to use graphic characters suitable for natural language. But some languages (e.g., French and Japanese) require graphic characters that are not present in ISO standard 646, thus making it difficult to write understandable comments in these languages and defeating the purpose of comments, namely, "the enlightenment of the human reader (2.7(1))."

2.7(1) defines the rules for comments as follows:

> A comment starts with two adjacent hyphens and extends up to the end of the line. [2.2(3) says the language does not define what causes the end of a line.] ... The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the effect of a program...

Since comments are not supposed to affect a program and since comments are supposed to make programs more understandable, it is desirable to allow complete freedom with respect to the characters that can occur in a comment. On the other hand, it is not reasonable to require implementations to accept arbitrary character set extensions, even if use of the extended characters is limited to comments. Allowing the choice of comment characters to be implementation-dependent need not impair the portability of a program, since when transporting a program, a comment can be deleted if it contains any extended character that is not acceptable to the new implementation.

[A related issue concerns the predefined type CHARACTER. Just as it is useful to have additional characters in comments, so it would be useful to have additional graphic characters in string literals. However, augmenting the definition of the CHARACTER type to support the ISO eight-bit Latin-1 character set, for example, would be a substantive change that cannot be supported at this time, although it deserves consideration when the Standard is revised (see AI-00420). Allowing the character set for comments to be extended is not a substantive change since comments have no effect on the legality or meaning of a program.]

Ada Commentary ai-00343-ra.wj downloaded on Wed May 11 18:50:04 EDT 1988

!standard 03.05.09 (10) 86-12-01 AI-00343/05
!class ramification 85-06-18
!status approved by WG9/AJPO 86-11-26
!status approved by Director, AJPO 86-11-26
!status approved by WG9/Ada Board 86-11-18
!status panel/committee-approved 86-08-07 (reviewed)
!status committee-approved (8-0-1) 86-05-12 (pending editorial review)
!status work-item 86-04-08
!status received 85-06-18
!references 83-00558, 83-00565
!topic Decimal fixed point representations

!summary 86-07-01

An implementation can use decimal or binary representations for fixed point values as long as all model numbers are represented exactly.

!question 86-04-15

Can an implementation represent a fixed point number in decimal form if small is not specified or if small is specified as a power of 10?

!response 86-09-05

An implementation can use decimal or binary representations for fixed point values as long as all model numbers are represented exactly, but using a decimal representation will not necessarily save space. For example, consider the declaration:

```
type FIX is delta 0.1 range -99.5 .. 99.5;
```

The model number mantissas consist of all integers in the range -2047 .. 2047 since (in the absence of a representation clause specifying FIX'SMALL), the value of small must be 0.0625 (3.5.9(5)). This means the model numbers cover the range -2047 * 0.0625 (which is -128.0 + 0.0625) through 128.0 - 0.0625. Now consider the following declarations:

```
A : FIX := FIX'LAST;
B : FIX := 0.5;
G : BOOLEAN := A + B > 98.0;
```

The sum, A + B, is 100.0, which belongs to the range of model numbers for FIX, so no exception can be raised when computing this sum even though the result lies outside the subtype FIX.

If the model numbers are represented in decimal notation, four digits are required to represent the required range of mantissa values. Even the mantissa for the model number 99.5 requires four decimal digits (since this mantissa value is 1592).

Now if a length clause specifying FIX'SMALL is given, the situation is somewhat different:

```
for FIX'SMALL use 0.1;
```

The model numbers in this case cover the range -(1023 * 0.1) .. 102.3, so it is still required that 99.5 + 0.5 be evaluated without raising NUMERIC_ERROR. This means computations using the predefined arithmetic operations must generally use four decimal digits to ensure NUMERIC_ERROR is not raised incorrectly. On the other hand, three decimal digits will suffice to hold stored values of type FIX since these values can never lie outside the range -99.5 .. 99.5.

If binary representation is used when FIX'SMALL is 0.1, the set of model numbers is the same, of course. The value 1023 requires 10 bits and the value 995 also requires 10 bits, so there is no difference between the size of stored values of subtype FIX and the minimum size needed for values of the base type.

In short, the answer to the question is, "Yes, decimal representation can be used for the model numbers if accuracy and range requirements are satisfied."

END

DATE

FILMED

DTIC

9-88