INTERNATIONAL STANDARD

ISO/IEC 11430

First edition
1994-12-01

Information technology — Programming languages — Generic package of elementary functions for Ada

Technologies de l'information — Langages de programmation — Ensemble générique de fonctions élémentaires pour l'Ada

Reference number
ISO/IEC 11430:1994(E)

Contents

Page

Foreword v

Introduction vi

1 Scope 1

2 Normative reference 1

3 Functions provided 1

4 Instantiations 2

5 Implementations 2

6 Exceptions 3

7 Arguments outside the range of safe numbers 4

8 Method of specification of functions 4

9 Domain definitions 4

10 Range definitions 5

11 Accuracy requirements 5

12 Overflow 6

13 Infinities 6

14 Underflow 7

15 Specifications of the functions 7

15.1 `SQRT` — Square root 8

15.2 `LOG` — Natural logarithm 8

15.3 `LOG` — Logarithm to an arbitrary base 9

15.4 EXP — Exponential function 10
15.5 "**" — Exponentiation operator 10
15.6 SIN — Trigonometric sine function, natural cycle (angle in radians) 11
15.7 SIN — Trigonometric sine function, arbitrary cycle (angle in arbitrary units) 12
15.8 COS — Trigonometric cosine function, natural cycle (angle in radians) 12
15.9 COS — Trigonometric cosine function, arbitrary cycle (angle in arbitrary units) 13
15.10 TAN — Trigonometric tangent function, natural cycle (angle in radians) 14
15.11 TAN — Trigonometric tangent function, arbitrary cycle (angle in arbitrary units) 14
15.12 COT — Trigonometric cotangent function, natural cycle (angle in radians) 15
15.13 COT — Trigonometric cotangent function, arbitrary cycle (angle in arbitrary units) 16
15.14 ARCSIN — Inverse trigonometric sine function, natural cycle (angle in radians) 16
15.15 ARCSIN — Inverse trigonometric sine function, arbitrary cycle (angle in arbitrary units) 17
15.16 ARCCOS — Inverse trigonometric cosine function, natural cycle (angle in radians) 18
15.17 ARCCOS — Inverse trigonometric cosine function, arbitrary cycle (angle in arbitrary units) 19
15.18 ARCTAN — Inverse trigonometric tangent function, natural cycle (angle in radians) 19
15.19 ARCTAN — Inverse trigonometric tangent function, arbitrary cycle (angle in arbitrary units) 21
15.20 ARCCOT — Inverse trigonometric cotangent function, natural cycle (angle in radians) 22
15.21 ARCCOT — Inverse trigonometric cotangent function, arbitrary cycle (angle in arbitrary units) 24
15.22 SINH — Hyperbolic sine function 25
15.23 COSH — Hyperbolic cosine function 26
15.24 TANH — Hyperbolic tangent function 27
15.25 COTH — Hyperbolic cotangent function 27
15.26 ARCSINH — Inverse hyperbolic sine function 28
15.27 ARCCOSH — Inverse hyperbolic cosine function 28
15.28 ARCTANH — Inverse hyperbolic tangent function 29
15.29 ARCCOTH — Inverse hyperbolic cotangent function 30

Annexes

A Ada specification for GENERIC_ELEMENTARY_FUNCTIONS 31

B Ada specification for ELEMENTARY_FUNCTIONS_EXCEPTIONS 32

C Rationale 33
C.1 History 33
C.2 Relationship to Ada 9X 34
C.3 Use of generics 34
C.4 Range constraints in the generic actual type 34
C.5 Functions included 37
C.6 Parameter names of the "**" operator 37
C.7 Units of angular measure 38

C.8 Optionality of the CYCLE and BASE parameters 38
C.9 Purposes and determination of the accuracy requirements 39
C.10 Rôle of the range definitions 41
C.11 Treatment of exceptional conditions 41
C.12 Underflow 43
C.13 0.0**0.0 44
C.14 Accommodation of portable implementations of GENERIC_ELEMENTARY_FUNCTIONS 44
C.15 Rôle of "signed zeros" and infinities 45
C.16 Mathematical constants 48

D Bibliography 49

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 11430 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 22, *Programming languages, their environments and system software interfaces.*

Annexes A and B form an integral part of this International Standard. Annexes C and D are for information only.

Introduction

The generic package described here is intended to provide the basic mathematical routines from which portable, reusable applications can be built. This International Standard serves a broad class of applications with reasonable ease of use, while demanding implementations that are of high quality, capable of validation and also practical given the state of the art.

The two specifications included in this International Standard are presented as compilable Ada specifications in annexes A and B with explanatory text in numbered clauses in the main body of text. The explanatory text is normative, with the exception of the following items:

— in clause 15, examples of common usage of the elementary functions (under the heading *Usage* associated with each function); and

— notes.

The word "may" as used in this International Standard consistently means "is allowed to" (or "are allowed to"). It is used only to express permission, as in the commonly occurring phrase "an implementation may"; other words (such as "can," "could" or "might") are used to express ability, possibility, capacity or consequentiality.

Information technology — Programming languages — Generic package of elementary functions for Ada

1 Scope

This International Standard defines the specification of a generic package of elementary functions called `GENERIC_ELEMENTARY_FUNCTIONS` and the specification of a package of related exceptions called `ELEMENTARY_FUNCTIONS_EXCEPTIONS`. It does not define the body of the former. No body is required for the latter.

This International Standard specifies certain elementary mathematical functions which are needed to support general floating-point usage and to support generic packages for complex arithmetic and complex functions. The functions were chosen because of their widespread utility in various application areas.

This International Standard is applicable to programming environments conforming to ISO 8652:1987.

2 Normative reference

The following standard contains provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the edition indicated was valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent edition of the standard indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 8652:1987, *Programming languages — Ada* (Endorsement of ANSI Standard 1815A-1983)

3 Functions provided

The following twenty mathematical functions are provided:

```
SQRT      LOG       EXP       "**"
SIN       COS       TAN       COT
ARCSIN    ARCCOS    ARCTAN    ARCCOT
SINH      COSH      TANH      COTH
ARCSINH   ARCCOSH   ARCTANH   ARCCOTH
```

These are the square root (`SQRT`), logarithm (`LOG`) and exponential (`EXP`) functions and the exponentiation operator (`**`); the trigonometric functions for sine (`SIN`), cosine (`COS`), tangent (`TAN`) and cotangent (`COT`) and their inverses (`ARCSIN`, `ARCCOS`, `ARCTAN` and `ARCCOT`); and the hyperbolic functions for sine (`SINH`), cosine (`COSH`), tangent (`TANH`) and cotangent (`COTH`) together with their inverses (`ARCSINH`, `ARCCOSH`, `ARCTANH` and `ARCCOTH`).

4 Instantiations

This International Standard describes a generic package, GENERIC_ELEMENTARY_FUNCTIONS, which the user must instantiate to obtain a package. It has one generic formal parameter, which is a generic formal type named FLOAT_TYPE. At instantiation, the user must specify a floating-point subtype as the generic actual parameter to be associated with FLOAT_TYPE; it is referred to below as the "generic actual type." This type is used as the parameter and result type of the functions contained in the generic package.

Depending on the implementation, the user may or may not be allowed to specify a generic actual type having a range constraint (see clause 5). If allowed, such a range constraint will have the usual effect of causing CONSTRAINT_ERROR to be raised when an argument outside the user's range is passed in a call to one of the functions, or when one of the functions attempts to return a value outside the user's range. Allowing the generic actual type to have a range constraint also has some implications for implementors.

In addition to the body of the generic package itself, implementors may provide (non-generic) library packages that can be used just like instantiations of the generic package for the predefined floating-point types. The name of a package serving as a replacement for an instantiation of GENERIC_ELEMENTARY_FUNCTIONS for the predefined type FLOAT should be ELEMENTARY_FUNCTIONS; for LONG_FLOAT and SHORT_FLOAT, the names should be LONG_ELEMENTARY_FUNCTIONS and SHORT_ELEMENTARY_FUNCTIONS, respectively; etc. When such a package is used in an application in lieu of an instantiation of GENERIC_ELEMENTARY_FUNCTIONS, it shall have the semantics implied by this International Standard for an instantiation of the generic package.

5 Implementations

Portable implementations of the body of GENERIC_ELEMENTARY_FUNCTIONS are strongly encouraged. However, implementations are not required to be portable. In particular, an implementation of this International Standard in Ada may use pragma INTERFACE or other pragmas, unchecked conversion, machine-code insertions or other machine-dependent techniques as desired.

An implementation may limit the precision it supports (by stating an assumed maximum value for SYSTEM.MAX_DIGITS), since portable implementations would not, in general, be possible otherwise. An implementation is also allowed to make other reasonable assumptions about the environment in which it is to be used, but only when necessary in order to match algorithms to hardware characteristics in an economical manner. All such limits and assumptions shall be clearly documented. By convention, an implementation of GENERIC_ELEMENTARY_FUNCTIONS is said not to conform to this International Standard in any environment in which its limits or assumptions are not satisfied, and the standard does not define its behavior in that environment. In effect, this convention delimits the portability of implementations.

An implementation may impose a restriction that the generic actual type shall not have a range constraint that reduces the range of allowable values. If it does impose this restriction, then the restriction shall be documented, and the effects of violating the restriction shall be one of the following:

— Compilation of a unit containing an instantiation of GENERIC_ELEMENTARY_FUNCTIONS is rejected.

— CONSTRAINT_ERROR or PROGRAM_ERROR is raised during the elaboration of an instantiation of GENERIC_ELEMENTARY_FUNCTIONS.

Conversely, if an implementation does not impose the restriction, then such a range constraint shall not be allowed, when included with the user's actual type, to interfere with the internal computations of the functions; that is, if the argument and result are within the range of the type, then the implementation shall return the result and shall not raise an exception (such as CONSTRAINT_ERROR).

An implementation shall function properly in a tasking environment. Apart from the obvious restriction that an implementation of GENERIC_ELEMENTARY_FUNCTIONS shall avoid declaring variables that are global to the functions, no special constraints are imposed on implementations. Nothing in this International Standard requires the use of such global variables.

Some hardware and their accompanying Ada implementations have the capability of representing and discriminating between positively and negatively signed zeros as a means (for example) of preserving the sign of an infinitesimal quantity that has underflowed to zero. Implementations of GENERIC_ELEMENTARY_FUNCTIONS may exploit that capability, when available, so as to exhibit continuity in the results of ARCTAN and ARCCOT as certain limits are approached. At the same time, implementations in which that capability is unavailable are also allowed. Because a definition of what comprises the capability of representing and distinguishing signed zeros is beyond the scope of this International Standard, implementations are allowed the freedom not to exploit the capability, even when it is available. This International Standard does not specify the sign that an implementation exploiting signed zeros shall give to a zero result; it does, however, specify that an implementation exploiting signed zeros shall yield results for ARCTAN and ARCCOT that depend on the sign of a zero argument. An implementation shall exercise its choice consistently, either exploiting signed-zero behavior everywhere or nowhere in this package. In addition, an implementation shall document its behavior with respect to signed zeros.

6 Exceptions

One exception, ARGUMENT_ERROR, is declared in GENERIC_ELEMENTARY_FUNCTIONS. This exception is raised by a function in the generic package only when the argument(s) of the function violate one or more of the conditions given in the function's domain definition (see clause 9).

NOTE — These conditions are related only to the mathematical definition of the function and are therefore implementation independent.

The ARGUMENT_ERROR exception is declared as a renaming of the exception of the same name declared in the ELEMENTARY_FUNCTIONS_EXCEPTIONS package. Thus, this exception distinguishes neither between different kinds of argument errors, nor between different functions, nor between different instantiations of GENERIC_ELEMENTARY_FUNCTIONS. Besides ARGUMENT_ERROR, the only exceptions allowed during a call to a function in GENERIC_ELEMENTARY_FUNCTIONS are predefined exceptions, as follows:

— Virtually any predefined exception is possible during the evaluation of an argument of a function in GENERIC_ELEMENTARY_FUNCTIONS. For example, NUMERIC_ERROR, CONSTRAINT_ERROR or even PROGRAM_ERROR could be raised if an argument has an undefined value; and, as stated in clause 4, if the implementation allows range constraints in the generic actual type, then CONSTRAINT_ERROR will be raised when the value of an argument lies outside the range of the user's generic actual type. Additionally, STORAGE_ERROR could be raised, e.g. if insufficient storage is available to perform the call. All these exceptions are raised before the body of the function is entered and therefore have no bearing on implementations of GENERIC_ELEMENTARY_FUNCTIONS.

— Also as stated in clause 4, if the implementation allows range constraints in the generic actual type, then CONSTRAINT_ERROR will be raised when a function in GENERIC_ELEMENTARY_FUNCTIONS attempts to return a value outside the range of the user's generic actual type. The exception raised for this reason shall be propagated to the caller of the function.

— Whenever the arguments of a function are such that a result permitted by the accuracy requirements would exceed FLOAT_TYPE'SAFE_LARGE in absolute value, as formalized below in clause 12, an implementation may raise (and shall then propagate to the caller) the exception specified by Ada for signaling overflow.

— Whenever the arguments of a function are such that the corresponding mathematical function is infinite (see clause 13), an implementation shall raise and propagate to the caller the exception specified by Ada for signaling division by zero.

— Once execution of the body of a function has begun, an implementation may propagate STORAGE_ERROR to the caller of the function, but only to signal the exhaustion of storage. Similarly, once execution of the body of a function has begun, an implementation may propagate PROGRAM_ERROR to the caller of the function, but only to signal errors made by the user of GENERIC_ELEMENTARY_FUNCTIONS.

No exception is allowed during a call to a function in GENERIC_ELEMENTARY_FUNCTIONS except those permitted by the foregoing rules. In particular, for arguments for which all results satisfying the accuracy requirements remain

less than or equal to FLOAT_TYPE'SAFE_LARGE in absolute value, a function shall handle locally an overflow occurring during the computation of an intermediate result, if such an overflow is possible, and shall not propagate an exception signaling that overflow to the caller of the function.

The only exceptions allowed during an instantiation of GENERIC_ELEMENTARY_FUNCTIONS, including the execution of the optional sequence of statements in the body of the instance, are CONSTRAINT_ERROR, PROGRAM_ERROR and STORAGE_ERROR, and then only for the reasons given below. The raising of CONSTRAINT_ERROR during instantiation is only allowed when the implementation imposes the restriction that the generic actual type shall not have a range constraint, and the user violates that restriction (it can, in fact, be an inescapable consequence of the violation). The raising of PROGRAM_ERROR during instantiation is only allowed for the purpose of signaling errors made by the user—for example, violation of this same restriction or of other limitations of the implementation. The raising of STORAGE_ERROR during instantiation is only allowed for the purpose of signaling the exhaustion of storage.

NOTE — In the Ada Reference Manual, the exception specified for signaling overflow or division by zero is NUMERIC_ERROR, but AI-00387 replaces that by CONSTRAINT_ERROR.

7 Arguments outside the range of safe numbers

ISO 8652:1987 fails to define the result safe interval of any basic or predefined operation of a real subtype when the absolute value of one of its operands exceeds the largest safe number of the operand subtype. (The failure to define a result in this case occurs because no safe interval is defined for the operand in question.) In order to avoid imposing requirements that would, consequently, be more stringent than those of Ada itself, this International Standard likewise does not define the result of a contained function when the absolute value of one of its arguments exceeds FLOAT_TYPE'SAFE_LARGE. All of the accuracy requirements and other provisions of the following clauses are understood to be implicitly qualified by the assumption that function arguments are less than or equal to FLOAT_TYPE'SAFE_LARGE in absolute value.

8 Method of specification of functions

Some of the functions have two overloaded forms. For each form of a function covered by this International Standard, the function is specified by its parameter and result type profile, the domain of its argument(s), its range and the accuracy required of its implementation. The meaning of, and conventions applicable to, the domain, range and accuracy specifications are described below.

9 Domain definitions

The specification of each function covered by this International Standard includes, under the heading *Domain*, a characterization of the argument values for which the function is mathematically defined. It is expressed by inequalities or other conditions which the arguments must satisfy to be valid. The phrase "mathematically unbounded" in a domain definition indicates that all representable values of the argument are valid. Whenever the arguments fail to satisfy all the conditions, the implementation shall raise ARGUMENT_ERROR. It shall not raise that exception if all the conditions are satisfied.

Inability to deliver a result for valid arguments (because the result overflows, for example) shall not raise ARGUMENT_ERROR, but shall be treated in the same way that Ada defines for its predefined floating-point operations (see clause 12).

NOTE — Unbounded portions of the domains of the functions EXP, "**", SINH and COSH, which are "expansion" functions with unbounded or semi-unbounded mathematical domains, are unexploitable because the corresponding function values (satisfying the accuracy requirements) cannot be represented. Their "usable domains," i.e. the portions of the mathematical domains given in their domain definitions that are exploitable in the sense that they produce representable results, are given by the notes accompanying their specifications. Because of permitted variations in implementations, these usable domains can only be stated approximately. In a similar manner, functions such as TAN and COT with periodic "poles" in their domains can (depending on the implementation) have small unusable portions of their domains in the vicinities of the poles. Also, range constraints in the user's generic actual type can, by narrowing a function's range, make further portions of the function's domain unusable.

10 Range definitions

The usual mathematical meaning of the "range" of a function is the set of values into which the function maps the values in its domain. Some of the functions covered by this International Standard (for example, `ARCSIN`) are mathematically multivalued, in the sense that a given argument value can be mapped by the function into many different result values. By means of range restrictions, this International Standard imposes a uniqueness requirement on the results of multivalued functions, thereby reducing them to single-valued functions.

Some of the functions covered by this International Standard (for example, `EXP`) have asymptotic behavior for extremely positive or negative arguments. Although there is no finite argument for which such a function can mathematically yield its asymptotic limit, that limit is always included in its range here, and it is an allowed result of the implemented function, in recognition of the fact that the limit value itself could be closer to the mathematical result than any other representable value.

The range of each function is shown under the heading *Range* in the specifications. Range definitions take the form of inequalities limiting the function value. An implementation shall not exceed a limit of the range when that limit is a safe number of `FLOAT_TYPE` (like `0.0`, `1.0` or `CYCLE/4.0` for certain values of `CYCLE`). On the other hand, when a range limit is not a safe number of `FLOAT_TYPE` (like π or `CYCLE/4.0` for certain other values of `CYCLE`), an implementation may exceed the range limit, but may not exceed the safe number of `FLOAT_TYPE` next beyond the range limit in the direction away from the interior of the range; this is in general the best that can be expected from a portable implementation. Effectively, therefore, range definitions have the added effect of imposing accuracy requirements on implementations above and beyond those presented under the heading *Accuracy* in the specifications (see clause 11).

The phrase "mathematically unbounded" in a range definition indicates that the range of values of the function is not bounded by its mathematical definition. It also implies that the function is not mathematically multivalued.

NOTE — Unbounded portions of the ranges of the functions `SQRT`, `LOG`, `ARCSINH` and `ARCCOSH`, which are "contraction" functions with unbounded or semi-unbounded mathematical ranges, are unreachable because the corresponding arguments cannot be represented. Their "reachable ranges," i.e. the portions of the mathematical ranges given in their range definitions that are reachable through appropriate arguments, are given by the notes accompanying their specifications. Because of permitted variations in implementations, these reachable ranges can only be stated approximately. Also, range constraints in the user's generic actual type can, by narrowing a function's domain, make further portions of the function's range unreachable.

11 Accuracy requirements

Because they are implemented on digital computers with only finite precision, the functions provided in this generic package can, at best, only approximate the corresponding mathematically defined functions.

The accuracy requirements contained in this International Standard define the latitude that implementations are allowed in approximating the intended precise mathematical result with floating-point computations. Accuracy requirements of two kinds are stated under the heading *Accuracy* in the specifications. Additionally, range definitions stated under the heading *Range* impose requirements that constrain the values implementations may yield, so the range definitions are another source of accuracy requirements (in that context, the precise meaning of a range limit that is not a safe number of `FLOAT_TYPE` is discussed in clause 10). Every result yielded by a function is subject to all of the function's applicable accuracy requirements, except in the one case described in clause 14. In that case, the result will satisfy a small absolute error requirement in lieu of the other accuracy requirements defined for the function.

The first kind of accuracy requirement used under the heading *Accuracy* in the specifications is a bound on the relative error in the computed value of the function, which shall hold (except as provided by the rules in clauses 12 and 14) for all arguments satisfying the conditions in the domain definition, providing the mathematical result is nonzero. For a given function f, the relative error $re(X)$ in a computed result $F(X)$ at the argument X is defined in the usual way,

$$re(X) = \left| \frac{F(X) - f(X)}{f(X)} \right|$$

providing the mathematical result $f(X)$ is finite and nonzero. (The relative error is not defined when the mathematical result is infinite or zero.) For each function, the bound on the relative error is identified under the heading *Accuracy* as its maximum relative error.

The second kind of accuracy requirement used under the heading *Accuracy* in the specifications is a stipulation, in the form of an equality, that the implementation shall deliver "prescribed results" for certain special arguments. It is used for two purposes:

— to define the computed result to be zero when the relative error is undefined, i.e., when the mathematical result is zero; and

— to strengthen the accuracy requirements at special argument values.

When such a prescribed result is a safe number of `FLOAT_TYPE` (like `0.0`, `1.0` or `CYCLE/4.0` for certain values of `CYCLE`), an implementation shall deliver that result. On the other hand, when a prescribed result is not a safe number of `FLOAT_TYPE` (like π or `CYCLE/4.0` for certain other values of `CYCLE`), an implementation may deliver any value in the surrounding safe interval. Prescribed results take precedence over maximum relative error requirements but never contravene them.

Range definitions, under the heading *Range* in the specifications, are an additional source of accuracy requirements, as stated in clause 10. As an accuracy requirement, a range definition (other than "mathematically unbounded") has the effect of eliminating some of the values permitted by the maximum relative error requirements, e.g. those outside the range.

12 Overflow

Floating-point hardware is typically incapable of representing numbers whose absolute value exceeds some implementation-defined maximum. For the type **`FLOAT_TYPE`**, that maximum will be at least **`FLOAT_TYPE'SAFE_LARGE`**. For the functions defined by this International Standard, whenever the maximum relative error requirements permit a result whose absolute value is greater than **`FLOAT_TYPE'SAFE_LARGE`**, the implementation may

— yield any result permitted by the maximum relative error requirements, or

— raise the exception specified by Ada for signaling overflow.

NOTES

1 The rule permits an implementation to raise an exception, instead of delivering a result, for arguments for which the mathematical result is close to but does not exceed **`FLOAT_TYPE'SAFE_LARGE`** in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result does exceed **`FLOAT_TYPE'SAFE_LARGE`** in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

2 The rule is motivated by the behavior prescribed by the Ada Reference Manual for the predefined operations. That is, when the set of possible results of a predefined operation includes a number whose absolute value exceeds the implementation-defined maximum, the implementation is allowed to raise the exception specified for signaling overflow instead of delivering a result.

3 In the Ada Reference Manual, the exception specified for signaling overflow is **`NUMERIC_ERROR`**, but AI-00387 replaces that by **`CONSTRAINT_ERROR`**.

13 Infinities

An implementation shall raise the exception specified by Ada for signaling division by zero in the following specific cases where the corresponding mathematical functions are infinite:

a) `LOG(X)` when `X = 0.0`;

b) `LOG(X, BASE)` when `X = 0.0`;

c) `LEFT ** RIGHT` when `LEFT = 0.0` and `RIGHT < 0.0`;

d) `TAN(X, CYCLE)` when `X` $= (2k + 1) \cdot$ `CYCLE/4.0`, for integer k;

e) `COT(X)` when `X` $=$ `0.0`;

f) `COT(X, CYCLE)` when `X` $= k \cdot$ `CYCLE/2.0`, for integer k;

g) `COTH(X)` when `X` $=$ `0.0`;

h) `ARCTANH(X)` when `X` $= \pm$`1.0`; and

i) `ARCCOTH(X)` when `X` $= \pm$`1.0`.

NOTE — In the Ada Reference Manual, the exception specified for signaling division by zero is `NUMERIC_ERROR`, but AI-00387 replaces that by `CONSTRAINT_ERROR`.

14 Underflow

Floating-point hardware is typically incapable of representing nonzero numbers whose absolute value is less than some implementation-defined minimum. For the type `FLOAT_TYPE`, that minimum will be at most `FLOAT_TYPE'SAFE_SMALL`. For the functions defined by this International Standard, whenever the maximum relative error requirements permit a result whose absolute value is less than `FLOAT_TYPE'SAFE_SMALL` and a prescribed result is not stipulated, the implementation may

a) yield any result permitted by the maximum relative error requirements;

b) yield any nonzero result having the correct sign and an absolute value less than or equal to `FLOAT_TYPE'SAFE_SMALL`; or

c) yield zero.

NOTES

1 Whenever the behavior on underflow is as described in 14 b) or 14 c), the maximum relative error requirements are, in general, unachievable and are waived. In such cases, the computed result will exhibit an error which, while not necessarily small in relative terms, is small in absolute terms. The absolute error will, in these cases, be less than or equal to `FLOAT_TYPE'SAFE_SMALL/(1.0−`*mre*), where *mre* is the maximum relative error specified for the function under the heading *Accuracy*.

2 The rule permits an implementation to deliver a result violating the maximum relative error requirements for arguments for which the mathematical result equals or slightly exceeds `FLOAT_TYPE'SAFE_SMALL` in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result is less than `FLOAT_TYPE'SAFE_SMALL` in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

3 The rule is motivated by the behavior prescribed by the Ada Reference Manual for predefined operations. That is, when the set of possible results of a predefined operation includes a nonzero number whose absolute value is less than the implementation-defined minimum, the implementation is allowed to yield zero or any nonzero number having the correct sign and an absolute value less than or equal to that minimum. An exception is never raised in this case.

15 Specifications of the functions

Under the heading *Definition* in each of the following specifications, the semantics of an Ada call to the function being defined is provided by a mathematical definition in the form of an approximation. The left-hand side (the function call) is set in the fixed-width font used throughout this International Standard for program fragments. The right-hand side is to be interpreted as an exact mathematical formula; as such, it and similar mathematical formulas throughout this International Standard employ standard mathematical symbols, notation and fonts (except for variable names and some real literals, which are set in the fixed-width "program-fragment" font). The degree to which the function call on the left-hand side is allowed to approximate the value of the formula on the right-hand side is, of course, spelled out under the heading *Accuracy*, as discussed in clause 11.

15.1 SQRT — Square root

15.1.1 Declaration

```
function SQRT (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.1.2 Definition

SQRT(X) $\approx \sqrt{X}$

15.1.3 Usage

```
Z := SQRT(X);
```

15.1.4 Domain

X $\geq$ 0.0

15.1.5 Range

SQRT(X) $\geq$ 0.0

NOTE — The upper bound of the reachable range of SQRT is approximately given by

$$\text{SQRT(X)} \leq \sqrt{\text{FLOAT\_TYPE'SAFE\_LARGE}}$$

15.1.6 Accuracy

a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON

b) SQRT(0.0) = 0.0

15.2 LOG — Natural logarithm

15.2.1 Declaration

```
function LOG (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.2.2 Definition

LOG(X) $\approx \log_e$ X

15.2.3 Usage

```
Z := LOG(X);    -- natural logarithm
```

15.2.4 Domain

X $\geq$ 0.0

NOTE — When X = 0.0, see clause 13.

15.2.5 Range

Mathematically unbounded

NOTE — The reachable range of LOG is approximately given by

$$\log_e \texttt{FLOAT\_TYPE'SAFE\_SMALL} \leq \texttt{LOG(X)} \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE}$$

15.2.6 Accuracy

a) Maximum relative error = $4.0 \cdot$ FLOAT_TYPE'BASE'EPSILON

b) LOG(1.0) = 0.0

15.3 LOG — Logarithm to an arbitrary base

15.3.1 Declaration

```
function LOG (X, BASE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.3.2 Definition

LOG(X, BASE) $\approx \log_{\texttt{BASE}}$ X

15.3.3 Usage

```
Z := LOG(X, 10.0);    -- base 10 logarithm
Z := LOG(X, 2.0);     -- base 2 logarithm
Z := LOG(X, BASE);    -- base BASE logarithm
```

15.3.4 Domain

a) X $\geq$ 0.0

b) BASE > 0.0

c) BASE $\neq$ 1.0

NOTE — When X = 0.0, see clause 13.

15.3.5 Range

Mathematically unbounded

NOTES

1 When BASE > 1.0, the reachable range of LOG is approximately given by

$$\log_{\texttt{BASE}} \texttt{FLOAT\_TYPE'SAFE\_SMALL} \leq \texttt{LOG(X, BASE)} \leq \log_{\texttt{BASE}} \texttt{FLOAT\_TYPE'SAFE\_LARGE}$$

2 When 0.0 < BASE < 1.0, the reachable range of LOG is approximately given by

$$\log_{\texttt{BASE}} \texttt{FLOAT\_TYPE'SAFE\_LARGE} \leq \texttt{LOG(X, BASE)} \leq \log_{\texttt{BASE}} \texttt{FLOAT\_TYPE'SAFE\_SMALL}$$

15.3.6 Accuracy

a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

b) LOG(1.0, BASE) = 0.0

15.4 EXP — Exponential function

15.4.1 Declaration

```
function EXP (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.4.2 Definition

EXP(X) ≈ e^X

15.4.3 Usage

```
Z := EXP(X);   -- e raised to the power X
```

15.4.4 Domain

Mathematically unbounded

NOTE — The usable domain of EXP is approximately given by

$$X \leq \log_e \text{FLOAT\_TYPE'SAFE\_LARGE}$$

15.4.5 Range

EXP(X) ≥ 0.0

15.4.6 Accuracy

a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

b) EXP(0.0) = 1.0

15.5 "**" — Exponentiation operator

15.5.1 Declaration

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

15.5.2 Definition

LEFT ** RIGHT ≈ $\text{LEFT}^{\text{RIGHT}}$

15.5.3 Usage

```
Z := LEFT ** RIGHT;   -- LEFT raised to the power RIGHT
```

15.5.4 Domain

a) LEFT $\geq$ 0.0

b) RIGHT $\neq$ 0.0 when LEFT = 0.0

NOTES

1 The usable domain of "**", when LEFT > 0.0, is approximately the set of values for LEFT and RIGHT satisfying

$$\text{RIGHT} \cdot \log_e \text{LEFT} \leq \log_e \text{FLOAT\_TYPE'SAFE\_LARGE}$$

This imposes a positive upper bound on RIGHT (as a function of LEFT) when LEFT > 1.0 and a negative lower bound on RIGHT (as a function of LEFT) when 0.0 < LEFT < 1.0.

2 When LEFT = 0.0 and RIGHT < 0.0 (together), see clause 13.

15.5.5 Range

LEFT ** RIGHT $\geq$ 0.0

15.5.6 Accuracy

a) Maximum relative error (when LEFT > 0.0) =

$$\left(4.0 + \frac{|\text{RIGHT} \cdot \log_e \text{LEFT}|}{32.0}\right) \cdot \text{FLOAT\_TYPE'BASE'EPSILON}$$

b) LEFT ** 0.0 = 1.0 when LEFT > 0.0

c) 0.0 ** RIGHT = 0.0 when RIGHT > 0.0

d) LEFT ** 1.0 = LEFT

e) 1.0 ** RIGHT = 1.0

15.6 SIN — Trigonometric sine function, natural cycle (angle in radians)

15.6.1 Declaration

```
function SIN (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.6.2 Definition

SIN(X) $\approx \sin$ X

15.6.3 Usage

```
Z := SIN(X);    -- X in radians
```

15.6.4 Domain

Mathematically unbounded

15.6.5 Range

$|\texttt{SIN(X)}| \leq \texttt{1.0}$

15.6.6 Accuracy

a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON when |X| is less than or equal to some documented implementation-dependent threshold, which shall not be less than

$$\texttt{FLOAT\_TYPE'MACHINE\_RADIX}^{\lfloor \texttt{FLOAT\_TYPE'MACHINE\_MANTISSA}/2 \rfloor}$$

For larger values of |X|, degraded accuracy is allowed. An implementation shall document its behavior for large |X|.

b) SIN(0.0) = 0.0

15.7 SIN — Trigonometric sine function, arbitrary cycle (angle in arbitrary units)

15.7.1 Declaration

```
function SIN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.7.2 Definition

$\texttt{SIN(X, CYCLE)} \approx \sin(2\pi \cdot \texttt{X}/\texttt{CYCLE})$

15.7.3 Usage

```
Z := SIN(X, 360.0);    -- X in degrees
Z := SIN(X, CYCLE);    -- X in units such that one complete cycle of rotation corresponds to
                       -- X = CYCLE
```

15.7.4 Domain

a) X mathematically unbounded

b) CYCLE > 0.0

15.7.5 Range

$|\texttt{SIN(X, CYCLE)}| \leq \texttt{1.0}$

15.7.6 Accuracy

a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON

b) For integer k, $\texttt{SIN(X, CYCLE)} = \begin{cases} 0.0, & \texttt{X} = k \cdot \texttt{CYCLE}/2.0 \\ 1.0, & \texttt{X} = (4k+1) \cdot \texttt{CYCLE}/4.0 \\ -1.0, & \texttt{X} = (4k+3) \cdot \texttt{CYCLE}/4.0 \end{cases}$

15.8 COS — Trigonometric cosine function, natural cycle (angle in radians)

15.8.1 Declaration

```
function COS (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.8.2 Definition

COS(X) ≈ cos X

15.8.3 Usage

```
Z := COS(X);    -- X in radians
```

15.8.4 Domain

Mathematically unbounded

15.8.5 Range

|COS(X)| ≤ 1.0

15.8.6 Accuracy

a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON when |X| is less than or equal to some documented implementation-dependent threshold, which shall not be less than

$$\text{FLOAT\_TYPE'MACHINE\_RADIX}^{\lfloor \text{FLOAT\_TYPE'MACHINE\_MANTISSA}/2 \rfloor}$$

For larger values of |X|, degraded accuracy is allowed. An implementation shall document its behavior for large |X|.

b) COS(0.0) = 1.0

15.9 COS — Trigonometric cosine function, arbitrary cycle (angle in arbitrary units)

15.9.1 Declaration

```
function COS (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.9.2 Definition

COS(X, CYCLE) ≈ cos(2π · X/CYCLE)

15.9.3 Usage

```
Z := COS(X, 360.0);    -- X in degrees
Z := COS(X, CYCLE);    -- X in units such that one complete cycle of rotation corresponds to
                       -- X = CYCLE
```

15.9.4 Domain

a) X mathematically unbounded

b) CYCLE > 0.0

15.9.5 Range

|COS(X, CYCLE)| ≤ 1.0

15.9.6 Accuracy

a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON

b) For integer k, COS(X, CYCLE) = $\begin{cases} 1.0, & X = k \cdot \text{CYCLE} \\ 0.0, & X = (2k+1) \cdot \text{CYCLE}/4.0 \\ -1.0, & X = (2k+1) \cdot \text{CYCLE}/2.0 \end{cases}$

15.10 TAN — Trigonometric tangent function, natural cycle (angle in radians)

15.10.1 Declaration

```
function TAN (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.10.2 Definition

TAN(X) ≈ tan X

15.10.3 Usage

```
Z := TAN(X);    -- X in radians
```

15.10.4 Domain

Mathematically unbounded

15.10.5 Range

Mathematically unbounded

15.10.6 Accuracy

a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON when |X| is less than or equal to some documented implementation-dependent threshold, which shall not be less than

$$\text{FLOAT\_TYPE'MACHINE\_RADIX}^{\lfloor \text{FLOAT\_TYPE'MACHINE\_MANTISSA}/2 \rfloor}$$

For larger values of |X|, degraded accuracy is allowed. An implementation shall document its behavior for large |X|.

b) TAN(0.0) = 0.0

15.11 TAN — Trigonometric tangent function, arbitrary cycle (angle in arbitrary units)

15.11.1 Declaration

```
function TAN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.11.2 Definition

TAN(X, CYCLE) ≈ tan(2π · X/CYCLE)

15.11.3 Usage

```
Z := TAN(X, 360.0);    -- X in degrees
Z := TAN(X, CYCLE);    -- X in units such that one complete cycle of rotation corresponds to
                       -- X = CYCLE
```

15.11.4 Domain

a) `X` mathematically unbounded

b) `CYCLE` > `0.0`

NOTE — When `X` = $(2k + 1) \cdot$ `CYCLE/4.0`, for integer k, see clause 13.

15.11.5 Range

Mathematically unbounded

15.11.6 Accuracy

a) Maximum relative error = `4.0` · `FLOAT_TYPE'BASE'EPSILON`

b) `TAN(X, CYCLE)` = `0.0` when `X` = $k \cdot$ `CYCLE/2.0`, for integer k

15.12 COT — Trigonometric cotangent function, natural cycle (angle in radians)

15.12.1 Declaration

```
function COT (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.12.2 Definition

`COT(X)` $\approx \cot$ `X`

15.12.3 Usage

```
Z := COT(X);    -- X in radians
```

15.12.4 Domain

Mathematically unbounded

NOTE — When `X` = `0.0`, see clause 13.

15.12.5 Range

Mathematically unbounded

15.12.6 Accuracy

Maximum relative error = `4.0` · `FLOAT_TYPE'BASE'EPSILON` when |`X`| is less than or equal to some documented implementation-dependent threshold, which shall not be less than

$$\texttt{FLOAT\_TYPE'MACHINE\_RADIX}^{\lfloor \texttt{FLOAT\_TYPE'MACHINE\_MANTISSA}/2 \rfloor}$$

For larger values of |`X`|, degraded accuracy is allowed. An implementation shall document its behavior for large |`X`|.

15.13 `COT` — Trigonometric cotangent function, arbitrary cycle (angle in arbitrary units)

15.13.1 Declaration

```
function COT (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.13.2 Definition

`COT(X, CYCLE)` ≈ $\cot(2\pi \cdot \texttt{X}/\texttt{CYCLE})$

15.13.3 Usage

```
Z := COT(X, 360.0);    -- X in degrees
Z := COT(X, CYCLE);    -- X in units such that one complete cycle of rotation corresponds to
                       -- X = CYCLE
```

15.13.4 Domain

a) `X` mathematically unbounded

b) `CYCLE` > `0.0`

NOTE — When `X` = k · `CYCLE/2.0`, for integer k, see clause 13.

15.13.5 Range

Mathematically unbounded

15.13.6 Accuracy

a) Maximum relative error = `4.0` · `FLOAT_TYPE'BASE'EPSILON`

b) `COT(X, CYCLE)` = `0.0` when `X` = $(2k+1)$ · `CYCLE/4.0`, for integer k

15.14 `ARCSIN` — Inverse trigonometric sine function, natural cycle (angle in radians)

15.14.1 Declaration

```
function ARCSIN (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.14.2 Definition

`ARCSIN(X)` ≈ $\arcsin$ `X`

15.14.3 Usage

```
Z := ARCSIN(X);    -- Z in radians
```

15.14.4 Domain

$|X| \leq 1.0$

15.14.5 Range

$|\text{ARCSIN}(X)| \leq \pi/2$

NOTE — $\pi/2$ and $-\pi/2$ are not safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits, but only slightly; see clause 10 for a precise statement of the requirements.

15.14.6 Accuracy

a) Maximum relative error = $4.0 \cdot$ FLOAT_TYPE'BASE'EPSILON

b) ARCSIN(0.0) = 0.0

c) ARCSIN(1.0) = $\pi/2$

d) ARCSIN(-1.0) = $-\pi/2$

NOTE — $\pi/2$ and $-\pi/2$ are not safe numbers of FLOAT_TYPE. Accordingly, when accuracy requirement 15.14.6 c) or 15.14.6 d) applies, an implementation may approximate the prescribed result, but only within narrow limits; see clause 11 for a precise statement of the requirements.

15.15 ARCSIN — Inverse trigonometric sine function, arbitrary cycle (angle in arbitrary units)

15.15.1 Declaration

```
function ARCSIN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.15.2 Definition

$\text{ARCSIN}(X, \text{CYCLE}) \approx (\arcsin X) \cdot \text{CYCLE}/2\pi$

15.15.3 Usage

```
Z := ARCSIN(X, 360.0);    -- Z in degrees
Z := ARCSIN(X, CYCLE);    -- Z in units such that one complete cycle of rotation corresponds to
                          -- Z = CYCLE
```

15.15.4 Domain

a) $|X| \leq 1.0$

b) CYCLE > 0.0

15.15.5 Range

$|\texttt{ARCSIN(X, CYCLE)}| \leq \texttt{CYCLE/4.0}$

NOTE — `CYCLE/4.0` and `−CYCLE/4.0` might not be safe numbers of `FLOAT_TYPE`. Accordingly, an implementation may exceed the range limits, but only slightly; see clause 10 for a precise statement of the requirements.

15.15.6 Accuracy

a) Maximum relative error = `4.0 · FLOAT_TYPE'BASE'EPSILON`

b) `ARCSIN(0.0, CYCLE) = 0.0`

c) `ARCSIN(1.0, CYCLE) = CYCLE/4.0`

d) `ARCSIN(-1.0, CYCLE) = −CYCLE/4.0`

NOTE — `CYCLE/4.0` and `−CYCLE/4.0` might not be safe numbers of `FLOAT_TYPE`. Accordingly, when accuracy requirement 15.15.6 c) or 15.15.6 d) applies, an implementation may approximate the prescribed result, but only within narrow limits; see clause 11 for a precise statement of the requirements.

15.16 ARCCOS — Inverse trigonometric cosine function, natural cycle (angle in radians)

15.16.1 Declaration

```
function ARCCOS (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.16.2 Definition

`ARCCOS(X)` ≈ arccos `X`

15.16.3 Usage

```
Z := ARCCOS(X);    -- Z in radians
```

15.16.4 Domain

$|\texttt{X}| \leq \texttt{1.0}$

15.16.5 Range

$\texttt{0.0} \leq \texttt{ARCCOS(X)} \leq \pi$

NOTE — π is not a safe number of `FLOAT_TYPE`. Accordingly, an implementation may exceed the upper range limit, but only slightly; see clause 10 for a precise statement of the requirements.

15.16.6 Accuracy

a) Maximum relative error = `4.0 · FLOAT_TYPE'BASE'EPSILON`

b) `ARCCOS(1.0) = 0.0`

c) `ARCCOS(0.0)` = $\pi/2$

d) `ARCCOS(-1.0)` = π

NOTE — $\pi/2$ and π are not safe numbers of `FLOAT_TYPE`. Accordingly, when accuracy requirement 15.16.6 c) or 15.16.6 d) applies, an implementation may approximate the prescribed result, but only within narrow limits; see clause 11 for a precise statement of the requirements.

15.17 ARCCOS — Inverse trigonometric cosine function, arbitrary cycle (angle in arbitrary units)

15.17.1 Declaration

```
function ARCCOS (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.17.2 Definition

ARCCOS(X, CYCLE) $\approx (\arccos X) \cdot$ CYCLE$/2\pi$

15.17.3 Usage

```
Z := ARCCOS(X, 360.0);    -- Z in degrees
Z := ARCCOS(X, CYCLE);    -- Z in units such that one complete cycle of rotation corresponds to
                          -- Z = CYCLE
```

15.17.4 Domain

a) $|X| \leq 1.0$

b) CYCLE > 0.0

15.17.5 Range

$0.0 \leq$ ARCCOS(X, CYCLE) $\leq$ CYCLE$/2.0$

NOTE — CYCLE/2.0 might not be a safe number of FLOAT_TYPE. Accordingly, an implementation may exceed the upper range limit, but only slightly; see clause 10 for a precise statement of the requirements.

15.17.6 Accuracy

a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

b) ARCCOS(1.0, CYCLE) = 0.0

c) ARCCOS(0.0, CYCLE) = CYCLE/4.0

d) ARCCOS(-1.0, CYCLE) = CYCLE/2.0

NOTE — CYCLE/4.0 and CYCLE/2.0 might not be safe numbers of FLOAT_TYPE. Accordingly, when accuracy requirement 15.17.6 c) or 15.17.6 d) applies, an implementation may approximate the prescribed result, but only within narrow limits; see clause 11 for a precise statement of the requirements.

15.18 ARCTAN — Inverse trigonometric tangent function, natural cycle (angle in radians)

15.18.1 Declaration

```
function ARCTAN (Y : FLOAT_TYPE; X : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
```

15.18.2 Definition

a) ARCTAN(Y) ≈ arctan Y

b) In an implementation exploiting signed zeros (see clause 5),

$$\texttt{ARCTAN(Y, X)} \approx \begin{cases} \arctan(\texttt{Y/X}), & \texttt{X} \geq 0.0 \\ (\arctan(\texttt{Y/X})) + \pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} > 0.0 \\ \pi, & \texttt{X} < 0.0 \text{ and Y a positively signed zero} \\ -\pi, & \texttt{X} < 0.0 \text{ and Y a negatively signed zero} \\ (\arctan(\texttt{Y/X})) - \pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} < 0.0 \end{cases}$$

In an implementation not exploiting signed zeros,

$$\texttt{ARCTAN(Y, X)} \approx \begin{cases} \arctan(\texttt{Y/X}), & \texttt{X} \geq 0.0 \\ (\arctan(\texttt{Y/X})) + \pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} > 0.0 \\ \pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} = 0.0 \\ (\arctan(\texttt{Y/X})) - \pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} < 0.0 \end{cases}$$

15.18.3 Usage

```
Z := ARCTAN(Y);      -- Z, in radians, is the angle (in the quadrant containing the point
                     -- (1.0,Y)) whose tangent is Y
Z := ARCTAN(Y, X);   -- Z, in radians, is the angle (in the quadrant containing the point
                     -- (X,Y)) whose tangent is Y/X
```

15.18.4 Domain

X ≠ 0.0 when Y = 0.0

15.18.5 Range

a) |ARCTAN(Y)| ≤ π/2

b) In an implementation exploiting signed zeros (see clause 5),

1) 0.0 ≤ ARCTAN(Y, X) ≤ π when Y > 0.0 or when Y is a positively signed zero

2) −π ≤ ARCTAN(Y, X) ≤ 0.0 when Y < 0.0 or when Y is a negatively signed zero

In an implementation not exploiting signed zeros,

1) 0.0 ≤ ARCTAN(Y, X) ≤ π when Y ≥ 0.0

2) −π ≤ ARCTAN(Y, X) ≤ 0.0 when Y < 0.0

NOTE — π and −π are not safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits that involve these numbers, but only slightly; see clause 10 for a precise statement of the requirements.

15.18.6 Accuracy

a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

b) ARCTAN(0.0) = 0.0

c) When Y is zero,

$$\texttt{ARCTAN(Y, X)} = \begin{cases} 0.0, & \texttt{X} > 0.0 \\ \pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} \text{ a positively signed zero} \\ -\pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} \text{ a negatively signed zero} \end{cases}$$

in an implementation exploiting signed zeros (see clause 5). In an implementation not exploiting signed zeros,

$$\texttt{ARCTAN(0.0, X)} = \begin{cases} 0.0, & \texttt{X} > 0.0 \\ \pi, & \texttt{X} < 0.0 \end{cases}$$

d) $\texttt{ARCTAN(Y, 0.0)} = \begin{cases} \pi/2, & \texttt{Y} > 0.0 \\ -\pi/2, & \texttt{Y} < 0.0 \end{cases}$

NOTE — π, $-\pi$, $\pi/2$ and $-\pi/2$ are not safe numbers of FLOAT_TYPE. Accordingly, when accuracy requirement 15.18.6 c) or 15.18.6 d) applies, an implementation may approximate a prescribed result that involves one of these numbers, but only within narrow limits; see clause 11 for a precise statement of the requirements.

15.19 ARCTAN — Inverse trigonometric tangent function, arbitrary cycle (angle in arbitrary units)

15.19.1 Declaration

```
function ARCTAN (Y : FLOAT_TYPE; X : FLOAT_TYPE := 1.0; CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.19.2 Definition

a) $\texttt{ARCTAN(Y, CYCLE => CYCLE)} \approx (\arctan \texttt{Y}) \cdot \texttt{CYCLE}/2\pi$

b) In an implementation exploiting signed zeros (see clause 5),

$$\texttt{ARCTAN(Y, X, CYCLE)} \approx \begin{cases} (\arctan(\texttt{Y}/\texttt{X})) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} \geq 0.0 \\ ((\arctan(\texttt{Y}/\texttt{X})) + \pi) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} > 0.0 \\ \texttt{CYCLE}/2.0, & \texttt{X} < 0.0 \text{ and } \texttt{Y} \text{ a positively signed zero} \\ -\texttt{CYCLE}/2.0, & \texttt{X} < 0.0 \text{ and } \texttt{Y} \text{ a negatively signed zero} \\ ((\arctan(\texttt{Y}/\texttt{X})) - \pi) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} < 0.0 \end{cases}$$

In an implementation not exploiting signed zeros,

$$\texttt{ARCTAN(Y, X, CYCLE)} \approx \begin{cases} (\arctan(\texttt{Y}/\texttt{X})) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} \geq 0.0 \\ ((\arctan(\texttt{Y}/\texttt{X})) + \pi) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} > 0.0 \\ \texttt{CYCLE}/2.0, & \texttt{X} < 0.0 \text{ and } \texttt{Y} = 0.0 \\ ((\arctan(\texttt{Y}/\texttt{X})) - \pi) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} < 0.0 \end{cases}$$

15.19.3 Usage

```
Z := ARCTAN(Y, CYCLE => 360.0);    -- Z, in degrees, is the angle (in the quadrant containing
                                   -- the point (1.0,Y)) whose tangent is Y
Z := ARCTAN(Y, CYCLE => CYCLE);    -- Z, in units such that one complete cycle of rotation
                                   -- corresponds to Z = CYCLE, is the angle (in the quadrant
                                   -- containing the point (1.0,Y)) whose tangent is Y
Z := ARCTAN(Y, X, 360.0);          -- Z, in degrees, is the angle (in the quadrant containing
                                   -- the point (X,Y)) whose tangent is Y/X
Z := ARCTAN(Y, X, CYCLE);          -- Z, in units such that one complete cycle of rotation
                                   -- corresponds to Z = CYCLE, is the angle (in the quadrant
                                   -- containing the point (X,Y)) whose tangent is Y/X
```

15.19.4 Domain

a) X ≠ 0.0 when Y = 0.0

b) CYCLE > 0.0

15.19.5 Range

a) |ARCTAN(Y, CYCLE => CYCLE)| ≤ CYCLE/4.0

b) In an implementation exploiting signed zeros (see clause 5),

1) 0.0 ≤ ARCTAN(Y, X, CYCLE) ≤ CYCLE/2.0 when Y > 0.0 or when Y is a positively signed zero

2) −CYCLE/2.0 ≤ ARCTAN(Y, X, CYCLE) ≤ 0.0 when Y < 0.0 or when Y is a negatively signed zero

In an implementation not exploiting signed zeros,

1) 0.0 ≤ ARCTAN(Y, X, CYCLE) ≤ CYCLE/2.0 when Y ≥ 0.0

2) −CYCLE/2.0 ≤ ARCTAN(Y, X, CYCLE) ≤ 0.0 when Y < 0.0

NOTE — CYCLE/2.0 and −CYCLE/2.0 might not be safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits that involve these numbers, but only slightly; see clause 10 for a precise statement of the requirements.

15.19.6 Accuracy

a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

b) ARCTAN(0.0, CYCLE => CYCLE) = 0.0

c) When Y is zero,

$$\text{ARCTAN(Y, X, CYCLE)} = \begin{cases} 0.0, & \text{X} > 0.0 \\ \text{CYCLE/2.0}, & \text{X} < 0.0 \text{ and Y a positively signed zero} \\ -\text{CYCLE/2.0}, & \text{X} < 0.0 \text{ and Y a negatively signed zero} \end{cases}$$

in an implementation exploiting signed zeros (see clause 5). In an implementation not exploiting signed zeros,

$$\text{ARCTAN(0.0, X, CYCLE)} = \begin{cases} 0.0, & \text{X} > 0.0 \\ \text{CYCLE/2.0}, & \text{X} < 0.0 \end{cases}$$

d) $\text{ARCTAN(Y, 0.0, CYCLE)} = \begin{cases} \text{CYCLE/4.0}, & \text{Y} > 0.0 \\ -\text{CYCLE/4.0}, & \text{Y} < 0.0 \end{cases}$

NOTE — CYCLE/2.0, −CYCLE/2.0, CYCLE/4.0 and −CYCLE/4.0 might not be safe numbers of FLOAT_TYPE. Accordingly, when accuracy requirement 15.19.6 c) or 15.19.6 d) applies, an implementation may approximate a prescribed result that involves one of these numbers, but only within narrow limits; see clause 11 for a precise statement of the requirements.

15.20 ARCCOT — Inverse trigonometric cotangent function, natural cycle (angle in radians)

15.20.1 Declaration

```
function ARCCOT (X : FLOAT_TYPE; Y : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
```

15.20.2 Definition

a) `ARCCOT(X)` $\approx$ arccot `X`

b) In an implementation exploiting signed zeros (see clause 5),

$$\texttt{ARCCOT(X, Y)} \approx \begin{cases} \text{arccot}(\texttt{X/Y}), & \texttt{Y} > \texttt{0.0} \\ \texttt{0.0}, & \texttt{Y a zero and X} > \texttt{0.0} \\ \pi, & \texttt{Y a positively signed zero and X} < \texttt{0.0} \\ -\pi, & \texttt{Y a negatively signed zero and X} < \texttt{0.0} \\ (\text{arccot}(\texttt{X/Y})) - \pi, & \texttt{Y} < \texttt{0.0} \end{cases}$$

In an implementation not exploiting signed zeros,

$$\texttt{ARCCOT(X, Y)} \approx \begin{cases} \text{arccot}(\texttt{X/Y}), & \texttt{Y} > \texttt{0.0} \\ \texttt{0.0}, & \texttt{Y} = \texttt{0.0 and X} > \texttt{0.0} \\ \pi, & \texttt{Y} = \texttt{0.0 and X} < \texttt{0.0} \\ (\text{arccot}(\texttt{X/Y})) - \pi, & \texttt{Y} < \texttt{0.0} \end{cases}$$

15.20.3 Usage

```
Z := ARCCOT(X);       -- Z, in radians, is the angle (in the quadrant containing the point
                      -- (X,1.0)) whose cotangent is X
Z := ARCCOT(X, Y);    -- Z, in radians, is the angle (in the quadrant containing the point
                      -- (X,Y)) whose cotangent is X/Y
```

15.20.4 Domain

`Y` $\neq$ `0.0` when `X` = `0.0`

15.20.5 Range

a) `0.0` $\leq$ `ARCCOT(X)` $\leq \pi$

b) In an implementation exploiting signed zeros (see clause 5),

1) `0.0` $\leq$ `ARCCOT(X, Y)` $\leq \pi$ when `Y` > `0.0` or when `Y` is a positively signed zero

2) $-\pi \leq$ `ARCCOT(X, Y)` $\leq$ `0.0` when `Y` < `0.0` or when `Y` is a negatively signed zero

In an implementation not exploiting signed zeros,

1) `0.0` $\leq$ `ARCCOT(X, Y)` $\leq \pi$ when `Y` $\geq$ `0.0`

2) $-\pi \leq$ `ARCCOT(X, Y)` $\leq$ `0.0` when `Y` < `0.0`

NOTE — π and $-\pi$ are not safe numbers of `FLOAT_TYPE`. Accordingly, an implementation may exceed the range limits that involve these numbers, but only slightly; see clause 10 for a precise statement of the requirements.

15.20.6 Accuracy

a) Maximum relative error = `4.0 * FLOAT_TYPE'BASE'EPSILON`

b) `ARCCOT(0.0)` = $\pi/2$

c) $\texttt{ARCCOT(0.0, Y)} = \begin{cases} \pi/2, & \texttt{Y} > 0.0 \\ -\pi/2, & \texttt{Y} < 0.0 \end{cases}$

d) When Y is zero,

$$\texttt{ARCCOT(X, Y)} = \begin{cases} 0.0, & \texttt{X} > 0.0 \\ \pi, & \texttt{X} < 0.0 \text{ and Y a positively signed zero} \\ -\pi, & \texttt{X} < 0.0 \text{ and Y a negatively signed zero} \end{cases}$$

in an implementation exploiting signed zeros (see clause 5). In an implementation not exploiting signed zeros,

$$\texttt{ARCCOT(X, 0.0)} = \begin{cases} 0.0, & \texttt{X} > 0.0 \\ \pi, & \texttt{X} < 0.0 \end{cases}$$

NOTE — π, $-\pi$, $\pi/2$ and $-\pi/2$ are not safe numbers of FLOAT_TYPE. Accordingly, when accuracy requirement 15.20.6 b), 15.20.6 c) or 15.20.6 d) applies, an implementation may approximate a prescribed result that involves one of these numbers, but only within narrow limits; see clause 11 for a precise statement of the requirements.

15.21 ARCCOT — Inverse trigonometric cotangent function, arbitrary cycle (angle in arbitrary units)

15.21.1 Declaration

```
function ARCCOT (X : FLOAT_TYPE; Y : FLOAT_TYPE := 1.0; CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

15.21.2 Definition

a) $\texttt{ARCCOT(X, CYCLE => CYCLE)} \approx (\text{arccot}\,\texttt{X}) \cdot \texttt{CYCLE}/2\pi$

b) In an implementation exploiting signed zeros (see clause 5),

$$\texttt{ARCCOT(X, Y, CYCLE)} \approx \begin{cases} (\text{arccot}(\texttt{X}/\texttt{Y})) \cdot \texttt{CYCLE}/2\pi, & \texttt{Y} > 0.0 \\ 0.0, & \texttt{Y} \text{ a zero and } \texttt{X} > 0.0 \\ \texttt{CYCLE}/2.0, & \texttt{Y} \text{ a positively signed zero and } \texttt{X} < 0.0 \\ -\texttt{CYCLE}/2.0, & \texttt{Y} \text{ a negatively signed zero and } \texttt{X} < 0.0 \\ ((\text{arccot}(\texttt{X}/\texttt{Y})) - \pi) \cdot \texttt{CYCLE}/2\pi, & \texttt{Y} < 0.0 \end{cases}$$

In an implementation not exploiting signed zeros,

$$\texttt{ARCCOT(X, Y, CYCLE)} \approx \begin{cases} (\text{arccot}(\texttt{X}/\texttt{Y})) \cdot \texttt{CYCLE}/2\pi, & \texttt{Y} > 0.0 \\ 0.0, & \texttt{Y} = 0.0 \text{ and } \texttt{X} > 0.0 \\ \texttt{CYCLE}/2.0, & \texttt{Y} = 0.0 \text{ and } \texttt{X} < 0.0 \\ ((\text{arccot}(\texttt{X}/\texttt{Y})) - \pi) \cdot \texttt{CYCLE}/2\pi, & \texttt{Y} < 0.0 \end{cases}$$

15.21.3 Usage

```
Z := ARCCOT(X, CYCLE => 360.0);    -- Z, in degrees, is the angle (in the quadrant containing
                                   -- the point (X,1.0)) whose cotangent is X
Z := ARCCOT(X, CYCLE => CYCLE);    -- Z, in units such that one complete cycle of rotation
                                   -- corresponds to Z = CYCLE, is the angle (in the quadrant
                                   -- containing the point (X,1.0)) whose cotangent is X
Z := ARCCOT(X, Y, 360.0);          -- Z, in degrees, is the angle (in the quadrant containing
                                   -- the point (X,Y)) whose cotangent is X/Y
Z := ARCCOT(X, Y, CYCLE);          -- Z, in units such that one complete cycle of rotation
                                   -- corresponds to Z = CYCLE, is the angle (in the quadrant
                                   --  containing the point (X,Y)) whose cotangent is X/Y
```

15.21.4 Domain

a) Y ≠ 0.0 when X = 0.0

b) CYCLE > 0.0

15.21.5 Range

a) 0.0 ≤ ARCCOT(X, CYCLE => CYCLE) ≤ CYCLE/2.0

b) In an implementation exploiting signed zeros (see clause 5),

1) 0.0 ≤ ARCCOT(X, Y, CYCLE) ≤ CYCLE/2.0 when Y > 0.0 or when Y is a positively signed zero

2) −CYCLE/2.0 ≤ ARCCOT(X, Y, CYCLE) ≤ 0.0 when Y < 0.0 or when Y is a negatively signed zero

In an implementation not exploiting signed zeros,

1) 0.0 ≤ ARCCOT(X, Y, CYCLE) ≤ CYCLE/2.0 when Y ≥ 0.0

2) −CYCLE/2.0 ≤ ARCCOT(X, Y, CYCLE) ≤ 0.0 when Y < 0.0

NOTE — CYCLE/2.0 and −CYCLE/2.0 might not be safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits that involve these numbers, but only slightly; see clause 10 for a precise statement of the requirements.

15.21.6 Accuracy

a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

b) ARCCOT(0.0, CYCLE => CYCLE) = CYCLE/4.0

c) $$\text{ARCCOT(0.0, Y, CYCLE)} = \begin{cases} \text{CYCLE/4.0}, & \text{Y} > 0.0 \\ -\text{CYCLE/4.0}, & \text{Y} < 0.0 \end{cases}$$

d) When Y is zero,

$$\text{ARCCOT(X, Y, CYCLE)} = \begin{cases} 0.0, & \text{X} > 0.0 \\ \text{CYCLE/2.0}, & \text{X} < 0.0 \text{ and Y a positively signed zero} \\ -\text{CYCLE/2.0}, & \text{X} < 0.0 \text{ and Y a negatively signed zero} \end{cases}$$

in an implementation exploiting signed zeros (see clause 5). In an implementation not exploiting signed zeros,

$$\text{ARCCOT(X, 0.0, CYCLE)} = \begin{cases} 0.0, & \text{X} > 0.0 \\ \text{CYCLE/2.0}, & \text{X} < 0.0 \end{cases}$$

NOTE — CYCLE/2.0, −CYCLE/2.0, CYCLE/4.0 and −CYCLE/4.0 might not be safe numbers of FLOAT_TYPE. Accordingly, when accuracy requirement 15.21.6 b), 15.21.6 c) or 15.21.6 d) applies, an implementation may approximate a prescribed result that involves one of these numbers, but only within narrow limits; see clause 11 for a precise statement of the requirements.

15.22 SINH — Hyperbolic sine function

15.22.1 Declaration

```
function SINH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.22.2 Definition

SINH(X) ≈ sinh X

15.22.3 Usage

```
Z := SINH(X);
```

15.22.4 Domain

Mathematically unbounded

NOTE — The usable domain of SINH is approximately given by

$$|X| \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE} + \log_e 2.0$$

15.22.5 Range

Mathematically unbounded

15.22.6 Accuracy

a) Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON

b) SINH(0.0) = 0.0

15.23 COSH — Hyperbolic cosine function

15.23.1 Declaration

```
function COSH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.23.2 Definition

COSH(X) ≈ cosh X

15.23.3 Usage

```
Z := COSH(X);
```

15.23.4 Domain

Mathematically unbounded

NOTE — The usable domain of COSH is approximately given by

$$|X| \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE} + \log_e 2.0$$

15.23.5 Range

COSH(X) ≥ 1.0

15.23.6 Accuracy

a) Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON

b) COSH(0.0) = 1.0

15.24 TANH — Hyperbolic tangent function

15.24.1 Declaration

```
function TANH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.24.2 Definition

TANH(X) ≈ tanh X

15.24.3 Usage

```
Z := TANH(X);
```

15.24.4 Domain

Mathematically unbounded

15.24.5 Range

|TANH(X)| ≤ 1.0

15.24.6 Accuracy

a) Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON

b) TANH(0.0) = 0.0

15.25 COTH — Hyperbolic cotangent function

15.25.1 Declaration

```
function COTH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.25.2 Definition

COTH(X) ≈ coth X

15.25.3 Usage

```
Z := COTH(X);
```

15.25.4 Domain

Mathematically unbounded

NOTE — When `X = 0.0`, see clause 13.

15.25.5 Range

$|\texttt{COTH(X)}| \geq 1.0$

15.25.6 Accuracy

Maximum relative error = `8.0 · FLOAT_TYPE'BASE'EPSILON`

15.26 ARCSINH — Inverse hyperbolic sine function

15.26.1 Declaration

```
function ARCSINH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.26.2 Definition

`ARCSINH(X)` ≈ arcsinh `X`

15.26.3 Usage

```
Z := ARCSINH(X);
```

15.26.4 Domain

Mathematically unbounded

15.26.5 Range

Mathematically unbounded

NOTE — The reachable range of `ARCSINH` is approximately given by

$$|\texttt{ARCSINH(X)}| \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE} + \log_e 2.0$$

15.26.6 Accuracy

a) Maximum relative error = `8.0 · FLOAT_TYPE'BASE'EPSILON`

b) `ARCSINH(0.0) = 0.0`

15.27 ARCCOSH — Inverse hyperbolic cosine function

15.27.1 Declaration

```
function ARCCOSH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.27.2 Definition

ARCCOSH(X) ≈ arccosh X

15.27.3 Usage

```
Z := ARCCOSH(X);
```

15.27.4 Domain

X ≥ 1.0

15.27.5 Range

ARCCOSH(X) ≥ 0.0

NOTE — The upper bound of the reachable range of ARCCOSH is approximately given by

$$\texttt{ARCCOSH(X)} \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE} + \log_e 2.0$$

15.27.6 Accuracy

a) Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON

b) ARCCOSH(1.0) = 0.0

15.28 ARCTANH — Inverse hyperbolic tangent function

15.28.1 Declaration

```
function ARCTANH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.28.2 Definition

ARCTANH(X) ≈ arctanh X

15.28.3 Usage

```
Z := ARCTANH(X);
```

15.28.4 Domain

|X| ≤ 1.0

NOTE — When X = ±1.0, see clause 13.

15.28.5 Range

Mathematically unbounded

15.28.6 Accuracy

a) Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON

b) ARCTANH(0.0) = 0.0

15.29 ARCCOTH — Inverse hyperbolic cotangent function

15.29.1 Declaration

```
function ARCCOTH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

15.29.2 Definition

ARCCOTH(X) ≈ arccoth X

15.29.3 Usage

```
Z := ARCCOTH(X);
```

15.29.4 Domain

|X| ≥ 1.0

NOTE — When X = ±1.0, see clause 13.

15.29.5 Range

Mathematically unbounded

15.29.6 Accuracy

Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON

Annex A
(normative)
Ada specification for GENERIC_ELEMENTARY_FUNCTIONS

```
with ELEMENTARY_FUNCTIONS_EXCEPTIONS;
generic
   type FLOAT_TYPE is digits <>;
package GENERIC_ELEMENTARY_FUNCTIONS is

   function SQRT    (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function LOG     (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function LOG     (X, BASE     : FLOAT_TYPE)        return FLOAT_TYPE;
   function EXP     (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function "**"    (LEFT, RIGHT : FLOAT_TYPE)        return FLOAT_TYPE;

   function SIN     (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function SIN     (X, CYCLE    : FLOAT_TYPE)        return FLOAT_TYPE;
   function COS     (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function COS     (X, CYCLE    : FLOAT_TYPE)        return FLOAT_TYPE;
   function TAN     (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function TAN     (X, CYCLE    : FLOAT_TYPE)        return FLOAT_TYPE;
   function COT     (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function COT     (X, CYCLE    : FLOAT_TYPE)        return FLOAT_TYPE;

   function ARCSIN  (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCSIN  (X, CYCLE    : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOS  (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOS  (X, CYCLE    : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCTAN  (Y           : FLOAT_TYPE;
                     X           : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
   function ARCTAN  (Y           : FLOAT_TYPE;
                     X           : FLOAT_TYPE := 1.0;
                     CYCLE       : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOT  (X           : FLOAT_TYPE;
                     Y           : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
   function ARCCOT  (X           : FLOAT_TYPE;
                     Y           : FLOAT_TYPE := 1.0;
                     CYCLE       : FLOAT_TYPE)        return FLOAT_TYPE;

   function SINH    (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function COSH    (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function TANH    (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function COTH    (X           : FLOAT_TYPE)        return FLOAT_TYPE;

   function ARCSINH (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOSH (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCTANH (X           : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOTH (X           : FLOAT_TYPE)        return FLOAT_TYPE;

   ARGUMENT_ERROR : exception renames ELEMENTARY_FUNCTIONS_EXCEPTIONS.ARGUMENT_ERROR;

end GENERIC_ELEMENTARY_FUNCTIONS;
```

Annex B
(normative)
Ada specification for ELEMENTARY_FUNCTIONS_EXCEPTIONS

```
package ELEMENTARY_FUNCTIONS_EXCEPTIONS is
   ARGUMENT_ERROR : exception;
end ELEMENTARY_FUNCTIONS_EXCEPTIONS;
```

Annex C
(informative)
Rationale

This annex, a revision of [8], outlines the history of ISO/IEC 11430 and provides a rationale for its features. While the annex primarily discusses the reasons why decisions were made during the development of this International Standard to include certain features, it also discusses the sometimes subtle reasons for *not* adopting obvious alternatives to those features. Additionally, it covers some of the less readily apparent implications of the decisions made during the development of this International Standard, which will be of particular interest to implementors.

C.1 History

The absence of predefined elementary functions from Ada has been one of the deterrents to the portability of scientific and engineering applications software written in that language. The need for such functions has been widely recognized, as evidenced by the support given to them by compiler vendors in the form of proprietary packages, as well as by several purveyors of libraries of mathematical software. While this has served the immediate needs of programmers within their own environments, it has done little to solve the broader problem of portability of applications software using the elementary functions. The reason, of course, is the lack of commonality among the different packages: they differ in the number of functions implemented, their names, their parameter profiles, the handling of exceptional conditions, and even the use (or avoidance) of genericity.

Instead of including predefined elementary functions in Ada, its authors gave the language the necessary general features for defining and collecting subprograms together into libraries (e.g., packages, generics and subprogram overloading), and for creating portable and efficient numerical software in particular (e.g., a model of floating-point arithmetic, and environmental enquiries in the form of attributes), and then left it to experienced numerical analysts to do what they are uniquely qualified to do: apply those features to the task of specifying and implementing high-quality libraries of mathematical software. Numerical analysts were already engaged in this work before Ada itself was standardized. The need for standards was recognized early, with several preliminary proposals [9, 21, 22] published between 1982 and 1987. Other seminal papers on the content, philosophy and implementation of scientific libraries in Ada were collected together in [10] in 1986.

People and papers came together beginning in about that same year to form committees with working documents. The standardization effort was assigned to JTC 1 as Project JTC1.22.10.02. The technical work was performed by the ACM SIGAda Numerics Working Group (referred to in this annex as "the committee") and the Ada-Europe Numerics Working Group; it was supported and encouraged in the United States by the Ada Joint Program Office and the Strategic Defense Initiative Office of the U.S. Department of Defense, and in Europe by the Commission of the European Communities. Interim reports on the work of the former committee were presented in 1987 at the International Conference on the Ada Programming Language [23], in 1988 at the Sandia Workshop on Ada in Real-Time and Scientific Environments [7, 25], and more recently in several tutorials and colloquia. Various drafts of the working papers that eventually resulted in this International Standard were circulated, and the response to them was enthusiastic.

This work was adopted by the WG 9 Numerics Rapporteur Group (NRG) in March of 1989 and presented to WG 9 as working draft 1.0 of a proposed standard [18]. WG 9 approved the proposal in June, 1989, subject to minor revisions. The revised proposal, working draft 1.1, was initially processed by SC 22 as a Technical Report in early 1990, leading to a few editorial changes, and subsequently registered as CD (Committee Draft) 11430 [14]. The committee took advantage of that opportunity for change to incorporate a minor technical improvement, which permits implementations having the capability of exploiting signed zeros (a feature of the IEEE standards for floating-point arithmetic [13, 3]) to do so in appropriate ways. At the same time, the Ada-Europe Numerics Working Group advocated a change stemming from the difficulty that some implementors encountered while trying to produce efficient implementations that are also portable. The second revision of the proposal, CD 11430.2 (also circulated as working draft 1.2), was endorsed by the NRG in December 1990, approved as Draft International Standard (DIS) 11430 by SC 22 [16], and then approved by JTC 1 in May of 1992. The final International Standard, to which this rationale applies, differs (other than in editorial ways) from the DIS approved by JTC 1 only in the substitution of the names

LEFT and RIGHT for X and Y as the parameters of the exponentiation operator and in the inclusion of "floor" brackets ($\lfloor \ldots \rfloor$) around the exponent in the formula giving the threshold beyond which the trigonometric functions are allowed to yield degraded accuracy.[1)]

C.2 Relationship to Ada 9X

This International Standard is specifically intended to be used with Ada 83, as defined by [26] and endorsed by ISO 8652:1987. A slightly revised version of GENERIC_ELEMENTARY_FUNCTIONS is proposed to be incorporated directly into Ada 9X (see [2, 15]). There, GENERIC_ELEMENTARY_FUNCTIONS is a child of ADA.NUMERICS. Other differences between this International Standard and GENERIC_ELEMENTARY_FUNCTIONS as proposed for Ada 9X are discussed in clauses C.4, C.6, C.9, C.15 and C.16.

C.3 Use of generics

The package construct is the obvious mechanism for encapsulating a functionally cohesive set of subprograms and their related exceptions, global data, etc. The facilities of TEXT_IO are made available through that mechanism, for example. Using a generic package instead of an ordinary package is appropriate, furthermore, when the facilities to be encapsulated need to be parameterized by some property of the application in which they are to be used. In view of the rules for parameter associations, the inability to anticipate which floating-point type (or types) the programmer will choose for the application dictates that the package containing the elementary functions be made generic on the type of their formal parameters and returned value. For that reason the elementary functions package is indeed generic. It has one generic formal parameter, a generic formal type named FLOAT_TYPE (see clause 4). An instantiation of the generic package with a floating-point subtype for the generic actual type produces a package containing elementary functions that can be invoked with an argument or arguments of that subtype.

C.4 Range constraints in the generic actual type

Until December 1990, the generic actual type was not restricted in any way, and in particular it could be a constrained subtype (i.e., have a range constraint). CD 11430.2 was changed at that time, however, to allow implementations to impose a restriction that the generic actual type must not contain a range constraint that reduces the range of allowable values. If implementations choose not to impose such a restriction, then they have an obligation to prevent a range constraint in the generic actual type from interfering with their ability to deliver a value, which constrains the available implementation strategies somewhat. So that users will know what to expect from any particular implementation, such a restriction must be documented if it is imposed.

This issue was debated at the time that the earliest draft was being formulated. The potential difficulty (for implementors) of allowing a range constraint in the generic actual type was recognized immediately. Since the rôle of the generic formal type is essentially to parameterize the "working precision" in the generic package, it is tempting for implementors to use the generic formal type, FLOAT_TYPE, as the type mark in the declarations of temporary variables and perhaps even of constants in the body of GENERIC_ELEMENTARY_FUNCTIONS. The obvious problem with this straightforward implementation strategy is that FLOAT_TYPE inherits any range constraint that the corresponding generic actual type happens to contain, and this could well invalidate assignments to temporary variables and initialization of constants in the body—even when both the argument in a particular function invocation and the final result (if it could only be computed) satisfy the range constraint. The user must accept the responsibility of being able to pass values into and out of an elementary function whose argument and result type are constrained, of course, since constraint checks are required by Ada in those contexts and nothing the implementor does in the body can avoid them. So, for example, if the user instantiates GENERIC_ELEMENTARY_FUNCTIONS with a type declared as "digits 6 range 3.0 .. 20.0," then there is no hope of asking for the square root of 25.0 (because the argument will be outside the range of SQRT's parameter type) or of 4.0 (because the result will be outside the range of SQRT's result type), and any attempt to do so must necessarily raise CONSTRAINT_ERROR. On the other hand, it was universally

[1)] The first of these differences is discussed later in clause C.6. The second, which involves a slight change in a more-or-less arbitrarily chosen threshold, was motivated by the complaints of some implementors that the original threshold was difficult to implement and by the realization that other implementors had read the proposed standard as if the floor brackets had been present all along. Both were approved by the NRG.

judged to be unacceptable for an implementation to raise CONSTRAINT_ERROR when the square root of, say, 16.0 is requested (with the same constrained generic actual type), since both the argument and the result are within the range of the parameter and result types. And yet, there is a good chance that a straightforward implementation—one that uses variables or constants of type FLOAT_TYPE—*will* raise CONSTRAINT_ERROR in this case and in other seemingly innocuous cases.[2] Thus, in agreeing to allow range constraints in the generic actual type, the committee made it clear in the earliest drafts that such "gratuitous" exceptions must be avoided by implementations. It did so only after concluding that suitable implementation techniques (ones that avoid the use of FLOAT_TYPE as a type mark in declarations) are available.

The committee strongly favored allowing range constraints, at first. It felt that users would not accept restrictions in their freedom to instantiate GENERIC_ELEMENTARY_FUNCTIONS with an arbitrary floating-point subtype. Indeed, earlier versions of this rationale declaimed that the interests of users outweighed those of implementors in settling the range constraints issue. The problems facing implementors who wish to allow range constraints would vanish if there were a way, in a generic body, of declaring a variable with the precision of a floating-point generic formal parameter but without its range constraints, if any. Declaring a variable to be of type "digits FLOAT_TYPE'BASE'DIGITS" will not suffice, because the expression in a floating accuracy definition is required to be static, and attributes of a generic formal parameter are not static. Declaring a variable to be of type FLOAT_TYPE'BASE comes to mind also, but this is invalid because the BASE attribute can be used (in Ada 83) only in a prefix for other attributes.

What implementation strategies *are* available to implementors who wish to allow range constraints?

One method is to represent each elementary function, at the highest level, by a "shell" that merely "dispatches" to a lower-level function supporting the required precision. The proper lower-level function is determined by querying FLOAT_TYPE'BASE'DIGITS in a case-statement whose choices test symbolically for membership in the range of precisions mapping into each of the available predefined floating-point types. Lower-level versions of each of the elementary functions can be provided for each predefined floating-point type by one instantiation of an inner generic package for each such type. Because the generic actual type used to instantiate this inner generic package is never constrained, the inner generic package of lower-level functions can use its generic formal parameter for the type of its working variables without the risk of violating range constraints.

A straightforward and often-used strategy, this method has two drawbacks:

— If the method is to be portable, the inner generic package must be designed to accommodate, in each elementary function, the entire range of precisions to be supported. The multiple instantiations of this inner package will then lead, with some Ada compilers, to multiple copies of the code applicable to a given precision, even though only one copy is logically required. Thus, for example, code to perform double-precision computations will be present in the instantiation for a double-precision predefined type as well as in the instantiation for a single-precision predefined type, even though the latter will never be called upon to perform those computations. Currently implemented optimizations do not, in general, attack this version of the dead-code removal problem.

— The method suffers from a lack of portability related to variations, from one implementation of Ada to the next, in the number and names of the available predefined floating-point types. As a supplement to this strategy, a technique due to Chebat can be used to extend portability to a fixed set of *potentially* predefined type names chosen by the implementor, even if some names in the set do not actually exist as predefined types of the Ada implementation; however, Chebat's technique will not pick up predefined type names outside the anticipated set.[3]

A second method, which completely avoids reliance on predefined type names, has a similar case-statement in the body of each elementary function, with the choices represented by constants and with each case containing a block-statement that declares working variables of the appropriate precision, given as a constant. In practice, the number of cases can be sharply reduced, as shown by Tang [24], by grouping a range of consecutive precisions together into each choice, with the breakpoints chosen with representative hardware in mind. (The precision used in the declaration of working

[2] One standard argument reduction strategy is to transform the argument to a value in the range 0.25 .. 1.0. Clearly, an attempt to store the transformed argument in a variable of type FLOAT_TYPE will violate the inherited range constraint.

[3] The essence of the "Chebat trick" is to declare a set of potentially predefined type names (excluding FLOAT, which of course *is* predefined)—like SHORT_SHORT_FLOAT, SHORT_FLOAT, LONG_FLOAT and LONG_LONG_FLOAT—in a dummy library package; it is suitable to declare them all as derived types with FLOAT as their parent. Then, by RM 8.4(5), in a program unit that "withs" and "uses" the dummy library package, a name from the set denotes the type declared in package STANDARD, if one exists there; otherwise, it denotes the type declared in the dummy package. In the latter case, the properties of the type are presumably irrelevant by design.

variables in each case then becomes the constant representing the upper bound of the range of precisions of the case's choice. Attention to several details not discussed here is required.)

This method, too, has two drawbacks:

— Excess precision may sometimes be used when not required—i.e., precision may be wasted—in order to keep the cases to a manageable number.

— Expensive compromises are required to fit some of the steps of the typical realization of an elementary function into the case structure. The approximation step—which is sandwiched between the argument reduction step and the result construction step—fits into the case structure well; each case not only provides the precision to be used in the declarations of its local working variables, but it also serves as the locus within which the chosen approximation scheme can be tailored to that precision (for example, by using the appropriate number of terms, with appropriate coefficients, in a polynomial approximation). On the other hand, the dependence of the argument reduction and result construction steps on the precision required is confined to the declarations of the working variables needed in those steps; the same argument reduction algorithm or result construction algorithm can be used in each of the cases. To avoid the needless duplication of code (differing only in the precision of variables), one is motivated to take the argument reduction and result construction steps out of all the cases and to place a common argument reduction step before the case-statement and a common result construction step after the case-statement. The only precision that suffices for the variables used in these steps is the maximum available precision. When that precision is more than is required (e.g., because a case corresponding to a low precision is selected), this tactic can produce an unfortunate performance degradation, especially in Ada implementations in which the highest precision available is simulated in software.

The magnitude of the performance penalty induced by the second method, which nevertheless is favored by some implementors because it is inherently more portable than the first, became apparent only late in the development of this International Standard, as implementation experience began to build. Aided by a growing suspicion that the desire to instantiate `GENERIC_ELEMENTARY_FUNCTIONS` with a constrained generic actual type may not be as realistic as once thought,[4)] the realization led to a reevaluation of the original decision on range constraints, and to a recommendation from the Ada-Europe Numerics Working Group that the decision be partially reversed by allowing an implementation to impose a restriction against range constraints if it so wishes. By imposing this restriction, an implementation can use `FLOAT_TYPE` directly as the type mark for all of its working variables; the only need for a case-statement is to select an approximation method tailored to the precision required, and the cases will furthermore not need to contain block-statements (because they will not need to declare case-dependent local variables).

A programmer who requires portability to all implementations must avoid instantiating `GENERIC_ELEMENTARY_FUNCTIONS` with a constrained generic actual type. That is an inconvenience, but it is hardly more than that. A program designed around a constrained application type or subtype named `APPLICATION_TYPE`, used both to declare variables and to instantiate a generic math package, can be systematically modified as follows to avoid instantiating `GENERIC_ELEMENTARY_FUNCTIONS` with a constrained generic actual type:

— Find the declaration of the type or subtype `APPLICATION_TYPE`.

— Change the declaration so that the type or subtype has a new name, say `ORIGINAL_TYPE`. In the modified program, `ORIGINAL_TYPE` will be used *only* as the source of properties for a new type, `BASE_TYPE`, and a subtype thereof called `APPLICATION_TYPE`.

— Declare the type `BASE_TYPE` as "`digits ORIGINAL_TYPE'BASE'DIGITS`." Instantiate `GENERIC_ELEMENTARY_FUNCTIONS` with `BASE_TYPE` instead of `APPLICATION_TYPE`.

— Introduce a new declaration for `APPLICATION_TYPE` as that of a subtype declared as "`BASE_TYPE range BASE_TYPE(ORIGINAL_TYPE'FIRST) .. BASE_TYPE(ORIGINAL_TYPE'LAST)`." Use `APPLICATION_TYPE` in the same ways as in the original program, except for the instantiation of `GENERIC_ELEMENTARY_FUNCTIONS`.

[4)] It is hard to imagine how a *single* application-determined range constraint can be suitably applied to the inputs and outputs of *all* the elementary functions without causing a constraint violation *somewhere* on a call or a return.

The foregoing idiom caters to the worst case, in which the base type of APPLICATION_TYPE in the original program is anonymous; obviously, if its name is known, that name can simply be used to instantiate GENERIC_ELEMENTARY_FUNCTIONS. Also, the foregoing idiom assumes that APPLICATION_TYPE in the original program is not a generic formal type; if it is, the same technique can be applied one level out.

Clause 5 implies that implementations imposing the restriction must behave predictably when the restriction is violated; it says that instantiation of GENERIC_ELEMENTARY_FUNCTIONS with a generic actual type containing a range constraint that reduces the allowable range of values, in violation of the restriction, must result either in the rejection of the compilation of a unit containing an instantiation of GENERIC_ELEMENTARY_FUNCTIONS or in the raising of CONSTRAINT_ERROR or PROGRAM_ERROR during the elaboration of an instantiation of GENERIC_ELEMENTARY_FUNCTIONS. The first two actions are consequences of the semantics of Ada and are beyond the implementor's ability to influence; if neither of these occurs first, the implementor can ensure that the last action takes place by coding the following in the statement-sequence of the body of GENERIC_ELEMENTARY_FUNCTIONS:

```
if FLOAT_TYPE'FIRST > FLOAT_TYPE'BASE'FIRST or
   FLOAT_TYPE'LAST  < FLOAT_TYPE'BASE'LAST  then
   raise PROGRAM_ERROR;
end if;
```

Finally, in Ada 9X it is proposed that one can use, for example, FLOAT_TYPE'BASE as a typemark in an object declaration, and therefore there is no reason to retain, in the version of GENERIC_ELEMENTARY_FUNCTIONS proposed for Ada 9X (see clause C.2), the provision allowing implementations to restrict the generic actual type. As proposed in Ada 9X [15], the generic actual type of GENERIC_ELEMENTARY_FUNCTIONS can have a range constraint, which must not affect the internal computations of the implementation.

C.5 Functions included

Nineteen functions and one operator are defined in the elementary functions package. These are SQRT; the two functions (EXP and LOG) and one operator ("**") of the exponential family; the four commonly encountered functions (SIN, COS, TAN and COT) of the trigonometric family; their inverses (ARCSIN, ARCCOS, ARCTAN and ARCCOT); the four commonly encountered functions (SINH, COSH, TANH and COTH) of the hyperbolic family; and their inverses (ARCSINH, ARCCOSH, ARCTANH and ARCCOTH). They were chosen because of their widespread utility in scientific and engineering applications. Actually, the trigonometric functions and their inverses are each represented by a pair of overloaded functions, with different numbers of parameters; the same is true of the LOG function. With overloadings included, a total of twenty-eight functions and one operator are defined in the elementary functions package.

C.6 Parameter names of the "**" operator

Bowing to common practice, as well as to the precedent set by other standards (such as the IEEE standards for floating-point arithmetic), the committee chose the name X for the formal parameter of the monadic functions and the names X and Y for those of ARCTAN and ARCCOT. For uniformity within GENERIC_ELEMENTARY_FUNCTIONS, the names X and Y were therefore also used originally for the formal parameters of the "**" operator. The decision to do so was also influenced by the desire to use short names in the accuracy requirements, which in some cases are expressed in terms of the values of the formal parameters.

This decision held through CD 11430.2. Subsequent to that, however, WG 9 recommended that all overloadings of predefined operators in Ada 9X use the parameter names LEFT and RIGHT, following the convention that had been established in Ada 83. This affected the "**" operator in the version of GENERIC_ELEMENTARY_FUNCTIONS proposed for Ada 9X. To avoid a gratuitous difference between the "**" operator in GENERIC_ELEMENTARY_FUNCTIONS as defined by this International Standard and that in the version of GENERIC_ELEMENTARY_FUNCTIONS proposed for inclusion in Ada 9X, the NRG decided late in the development of this International Standard to use LEFT and RIGHT for the parameters of the "**" operator.

The overloading of "**" contained in GENERIC_ELEMENTARY_FUNCTIONS does differ from the obvious extension of the predefined "**" operator of package STANDARD, but not with respect to the names of its parameters (see clause C.13).

C.7 Units of angular measure

Users have a choice of units in which angles are measured, and the overloadings in the trigonometric family and their inverses play a rôle in the exercise of that choice. Most often, the desired angular measure is radians; consequently, it is the easiest to specify. Thus, `SIN(X)`, for example, yields the sine of the angle `X`, where `X` is understood to be measured in radians, and similarly `ARCSIN(X)` yields the angle (in radians) whose sine is `X`. To specify some other angular measure, one would supply a value for `CYCLE` (which is the second parameter, except in the case of `ARCTAN` or `ARCCOT`, where it is the third). For example, when the angle `X` is understood to be measured in degrees, the sine of `X` would be written instead as `SIN(X, 360.0)`; by the same token, the angle (in degrees) whose sine is `X` is `ARCSIN(X, 360.0)`. Other angular measures can be accommodated by using the appropriate value for `CYCLE`—e.g., `6400.0` for mils, `400.0` for grads. From these examples it should be clear that the numerical value of `CYCLE` has the following interpretation: an angle numerically equal to the value of `CYCLE` represents one complete cycle of revolution (i.e., one period of the function).[5] It should also be clear that when the `CYCLE` parameter is omitted, as in `SIN(X)`, the effect is as if a `CYCLE` of 2π had been specified.

A similar choice exists with respect to the base of the `LOG` function. `LOG(X)` means the natural or Napierian logarithm (i.e., base e); for other bases, such as `2.0` or `10.0`, which are perhaps the most common after e, one writes `LOG(X, 2.0)`, `LOG(X, 10.0)`, etc. The optional parameter is called `BASE`, and when it is omitted the effect is as if a `BASE` of e had been specified. There is no `BASE` parameter for the `EXP` function, which is the inverse of `LOG`, since that functionality can be obtained with the exponentiation operator: the number whose logarithm to the base `B` is `X` can be computed as `B**X`.

C.8 Optionality of the CYCLE and BASE parameters

The preceding discussion suggests that the optionality of the `CYCLE` and `BASE` parameters could have been handled very simply by defining appropriate default values for these parameters, and yet subprogram overloading was used instead, which unfortunately adds nine subprogram declarations to the specification of `GENERIC_ELEMENTARY_FUNCTIONS`. What is so disadvantageous about the default-value method to warrant this increase in the size of the specification of `GENERIC_ELEMENTARY_FUNCTIONS`? The answer is rather subtle, and it required much debate during the development of this International Standard. The problem is essentially that 2π and e, being irrational, are not representable exactly in any implementation, so the best that could possibly be done would be to use a default value that is a close approximation to 2π or e (implicit conversion of 2π or e, expressed as a numeric literal with an arbitrarily large number of trailing digits, to the type `FLOAT_TYPE` is only required to yield a value in the same safe interval as the literal). The use of such an approximation to 2π in the computation of the trigonometric functions, as if it were the true period, would produce results with unacceptable accuracy. The error is not like a simple roundoff error but has the nature of a cumulative phase shift that increases as the number of periods, or cycles, represented by `X` increases. For sufficiently large `X`, there will be no correct digits whatsoever in the result.

Better results can be obtained, and this problem ameliorated somewhat, if the implementation uses an internal representation of 2π that has more precision than `FLOAT_TYPE` provides. The necessary additional precision may be obtained from hardware, if it is available; if not, there are techniques for "simulating" extra precision (see, e.g., [24]). But using extra precision only pushes the "phase shift problem," wherein all accuracy is lost, farther away—i.e., to larger values of `X`; it does not eliminate it entirely. Therefore, the amount of extra precision required is related to the range of values of `X` for which some stated accuracy is to be achieved. Since the required accuracy and the minimum domain over which it must be achieved are both spelled out in this International Standard, implementors have the information they need to satisfy the standard's requirements. (Outside the range of `X` for which the trigonometric functions with natural cycle must meet the accuracy requirements, degraded accuracy—but not other behavior, such as the raising of an exception—is allowed. An implementation must document the accuracy it achieves for extreme values of `X`, along with the threshold where degradation below the required accuracy commences.)

[5] Early drafts of the proposals leading to this International Standard contained examples involving a `CYCLE` of `1.0`, allegedly corresponding to angular measure in bams ("binary angular measure"). It was tardily discovered [11] that this obscure invention is primarily useful in *fixed-point* implementations, where the primary range of the angle is taken to be $[-1, 1)$ bams, corresponding to $[-\pi, \pi)$ radians, and scaled to make use of all the bits in an integer word—for example, scaled to $[-32768, 32767]$. Thus, a full cycle in bams is `2.0`, not `1.0`. Since a `CYCLE` of `2.0` does not begin to suggest the real utility of bams, the examples involving bams were omitted after that discovery, instead of being corrected.

It turns out, in fact, that if the default-value method had been employed for CYCLE in this International Standard, and an implementation used that value as if it were the true period, then the range of values of X for which the stated accuracy could be achieved would not even extend as far as one complete period away from the origin. This also means that an implementation that calculates SIN(X) by calling SIN(X, P) for some value of P meant to approximate 2π, including a literal with an arbitrarily large number of digits, will fail to meet the specifications.

All of the other standard periods are given by CYCLE values that are representable on all machines. With the aid of an appropriate "exact-remainder" algorithm, implementations of the explicit-cycle forms of the functions will have no difficulty reducing arbitrarily large values of X to the primary interval near the origin without error. (In fact, the exact-remainder function is included in a generic package of floating-point manipulation functions called GENERIC_PRIMITIVE_FUNCTIONS which is also progressing as an International Standard; see [17].) That is why the explicit-cycle forms of the trigonometric functions have no restrictions on the range of values of X for which the stated accuracy requirements must hold.

It might have been reasonable to employ the default-value method for CYCLE and expect implementations to meet the accuracy requirements over the given range of values for X by recognizing when CYCLE has the default value but not using it (in that case) as if it were the true period—i.e., by using the default value merely as an indicator that a different argument-reduction technique should be used. This was actually considered but ultimately abandoned because of the possibility of non-monotonic behavior as CYCLE sweeps through the default value, and because of potential surprises that could occur should users supply an explicit CYCLE that they believe to be a close approximation to 2π but that is not identical to the default value. Making CYCLE an enumeration type was also considered, but that, too, was abandoned, largely because it would have limited the available periods to a fixed set; furthermore, obtaining a numerical value of the period denoted by the enumeration value represents an unnecessary overhead. Various other solutions of "the CYCLE problem" were also investigated, such as packaging the trigonometric functions and their inverses in an inner generic package having CYCLE as a generic formal parameter (e.g., a generic formal object of mode "in"). While these offered some elegant advantages, they also had unacceptable disadvantages.

The inverses of the trigonometric functions do not exhibit the phase shift phenomenon with respect to errors in their periods. For them, it is sufficient to compute the result as (for example) a fraction of a period and then scale that by multiplying by CYCLE. The default-value method for handling the optionality of CYCLE would have posed no problems, but subprogram overloading was preferred simply for reasons of uniformity. The same reasoning applies to BASE, in the case of LOG.

C.9 Purposes and determination of the accuracy requirements

One of the significant advances represented by this International Standard is its inclusion of accuracy requirements for implementations of the elementary functions. Not usually considered in formal specifications of mathematical software, accuracy requirements are made possible largely by the model (adapted from Brown [4]) of real arithmetic incorporated into Ada, and the form they take is influenced by the availability of attributes that characterize properties of Ada implementations relative to that model. Because the accuracy requirements will have an effect on what implementors can do, they will translate into some assurance of quality for the user; in addition, they will permit users to carry out error analyses of programs containing references to the elementary functions.

Two kinds of accuracy requirements—maximum-relative-error bounds and "prescribed results"—are included. All of the functions have maximum-relative-error bounds that limit the relative error in the computed result, over the whole range of valid arguments (or, in some cases, over a stated portion of the range). In addition, the results at certain key argument values are prescribed more precisely for some of the functions. The maximum-relative-error bounds were determined by numerical analysts having broad knowledge of algorithms and implementation techniques for the elementary functions. They are, of course, tailored to the specific properties of each function ([1, 5, 12] were of general help in this regard). They are considered to be realistic and to give implementors some leeway for creativity and individualism in regard to the trade-off between accuracy and efficiency. While they do rule out naïve implementations, they have proven to be conservative in the sense that it is not especially difficult for a knowledgeable implementor to produce implementations exceeding the accuracy requirements.

The maximum-relative-error bounds are based on the implemented precision of the generic actual parameter associated with FLOAT_TYPE rather than on its declared precision. This is reflected in the use of FLOAT_TYPE'BASE'EPSILON,

rather than FLOAT_TYPE'EPSILON, in the formulas for maximum relative error. Effectively, those formulas constrain the computed result to lie within an appropriate number of safe intervals of the mathematical result, just as Ada does for the predefined arithmetic operators, which is what motivated the use of the BASE attribute in these formulas. In practice, it means that the approximation technique employed by the implementation must be appropriate for the precision of the base type of FLOAT_TYPE rather than just that of FLOAT_TYPE itself—i.e., it must be capable of exploiting all the precision inherent in the underlying base type.[6)]

Error analyses of programs containing the elementary functions, using the maximum-relative-error bounds as given in the specifications of GENERIC_ELEMENTARY_FUNCTIONS, are qualitatively portable in the sense that their form does not change from one Ada implementation to another. They are not quantitatively portable, however, since the numerical size of the maximum-relative-error bounds depends on the Ada implementation's mapping between user-declared floating-point subtypes and the predefined floating-point types. Nevertheless, a quantitatively portable error analysis can also be carried out merely by substituting FLOAT_TYPE'EPSILON for FLOAT_TYPE'BASE'EPSILON wherever it arises in the analysis. The choice is equivalent to carrying out the error analysis either at the level of safe numbers or at the level of model numbers; both are qualitatively portable, but only the latter is quantitatively portable. Ada gives analysts that choice when they can "see" all the way down to the level of the basic operations and predefined operators in their Ada programs; this International Standard preserves that choice—without requiring one to look inside implementations of the elementary functions—by constraining and describing their behavior at the level of safe numbers (from which their behavior at the level of model numbers can be trivially inferred).

A considerable amount of debate was necessary to reach consensus on this form of the maximum-relative-error bounds. Some of the contributors to this International Standard felt that an implementation should be permitted to use coarser and coarser approximation methods as the precision of FLOAT_TYPE decreases (e.g., in different instantiations), even when the precision of its base type remains the same. For example, a graphics application might well not need 6-digit accuracy in the elementary functions when the user's generic actual subtype is declared as "digits 2," and the user might not be willing to pay for the unneeded accuracy in the form of additional iterations through some loop, or additional terms in an approximating polynomial, inside the body of an elementary function. A majority of the contributors felt that it was better to require software to get the most out of the hardware it is given to work with, at least for conforming implementations, reasoning that special requirements can always be met by additional implementations not conforming to the standard. There was also a question as to whether the use of the BASE attribute in the accuracy requirements adequately reflects the *implemented* precision of the user's generic actual subtype (for example, in the case of a reduced-accuracy subtype, perhaps combined with the influence of representation clauses). Ada Commentary AI-00407 [19] implies that the implemented precision of a reduced-accuracy subtype, as it affects the storage of variables of the subtype as well as parameter associations and function returns involving the subtype, may be less than the precision of the subtype's base type. Because that decision has profoundly undesirable consequences (including the obfuscation of the concept of "representation of a type"; the loss of the ability to specify and analyze the behavior of composite operations, represented by functions, using the same abstractions—including safe intervals—as are applicable to the basic and predefined operations; and the rendering of certain classes of attributes nearly useless), and because it appears to conflict with other requirements or implications of the language [26], some observers feel that it is ill advised. Accordingly, Ada Commentary AI-00571, which calls for the reevaluation of AI-00407, was submitted in July of 1988. WG 9 returned AI-00407 to the Ada Rapporteur Group for reconsideration in June of 1989, and the ARG completed its reconsideration four months later by approving AI-00571 (thereby rescinding AI-00407). Thus, it is now known that the accuracy laboriously achieved in the body of an elementary function will not be thrown away at the return, even when the generic actual subtype is a reduced-accuracy subtype.

Prescribed results are used in some cases to constrain the computed result even more than the maximum-relative-error bounds constrain it. For example, EXP(0.0) is prescribed to yield exactly 1.0. The prescribed results reflect behavior that is both highly desirable from a numerical point of view and easy to achieve. In most cases, sensible algorithms will achieve the required behavior without extra effort; when necessary, it can always be achieved with a test for the special argument values.[7)]

Some of the prescribed results appear to require the function to deliver a value that cannot be computed exactly; for

[6)] In the version of GENERIC_ELEMENTARY_FUNCTIONS proposed for Ada 9X, the maximum relative error is specified in terms of FLOAT_TYPE'MODEL_EPSILON. The MODEL_EPSILON attribute replaces EPSILON, but it yields a property of the base type of its prefix.

[7)] Another minor difference between this International Standard and the version of GENERIC_ELEMENTARY_FUNCTIONS proposed for Ada 9X is the inclusion of a prescribed result for SQRT(1.0) in the latter. The guarantee of exactness for SQRT(1.0) makes it realistic to prescribe exact results for some of the *complex* elementary functions (for which an International Standard is also under development) when their arguments lie on the real axis or the imaginary axis.

example, one of the prescribed results reads " `ARCSIN(1.0)` = $\pi/2$." What does this mean? Clause 11 says that a prescribed result that is a safe number must be delivered exactly; in the case of one that is not, such as this one, the implementation may deliver any value in the surrounding safe interval. The required behavior can be achieved without difficulty, even in portable implementations.

Consideration was given to allowing the trigonometric functions with natural cycle to raise `ARGUMENT_ERROR` for sufficiently large `X`, where it is impractical to avoid accuracy degradation. However, this would have weakened the significance of `ARGUMENT_ERROR` as an implementation-independent indicator of mathematical domain violation. Raising a different exception was also considered, but in the end the committee thought that most users would prefer continuation with a result that falls short of the accuracy requirements by a known amount to no result at all.

C.10 Rôle of the range definitions

Range definitions (or restrictions) are included with some of the functions for several reasons. In the case of functions that are mathematically multivalued, they serve to define the principal range for the implementation, enabling it to be single-valued without ambiguity. In other cases, they impose highly desirable and easily achieved numerical constraints on the results—constraints that do not automatically follow from the maximum-relative-error requirements. In this latter context, they behave like additional prescribed results (in the form of an inequality, rather than an equality). And, like prescribed results, range limits are sometimes given by values that cannot be computed exactly. In analogy to the meaning of prescribed results, clause 10 defines the meaning of range limits like this: When a range limit is a safe number, the implementation must not exceed the range limit; when it is not, the implementation may exceed it, but the implementation may not exceed the next safe number beyond the range limit in the direction away from the interior of the range. The required behavior can be achieved without difficulty, even in portable implementations. (For more on range definitions, see clause C.12.)

C.11 Treatment of exceptional conditions

Two types of exceptional condition are explicitly recognized by this International Standard; in each case, the defined action is to raise an exception. Equally important, an implementation is prohibited from raising spurious exceptions if it is to conform (see clause 6).

The first type of exceptional condition under which an implementation is allowed to raise an exception instead of delivering a result occurs when the arguments of one of the elementary functions are such that its mathematical result is not defined—in other words, when its arguments are invalid. A familiar example, given that arguments and results in `GENERIC_ELEMENTARY_FUNCTIONS` are restricted to the real domain (as opposed to the complex domain), is an attempt to compute the square root of a negative number. The validity or invalidity of arguments is completely defined by the "domain definitions" included with the description of each function.

When faced with invalid arguments, an implementation is not merely allowed to raise an exception; it is required to do so. This International Standard prescribes the raising of the `ARGUMENT_ERROR` exception, which it defines and reserves for this situation. The validity of given arguments is never influenced by hardware properties; if `ARGUMENT_ERROR` is raised by a function for certain arguments in one implementation, it will be raised for those arguments in any implementation. Argument validity can be reliably established by inspection of the arguments, that is, by subjecting them to appropriate tests. While it will usually be most convenient for an implementation to check for argument validity before attempting to compute a result, other strategies may be possible and appropriate in some cases.

The second type of exceptional condition under which an implementation is allowed to raise an exception instead of delivering a result occurs when the mathematical result is well defined for the given arguments but some exigency (from among a limited list) unavoidably stands in the way of actually delivering that result. There are in fact three misfortunes that can befall an implementation, interfering with its ability to deliver a numerical result that is close enough to the mathematical result to satisfy the accuracy requirements:

— It may happen that the given arguments fail to satisfy range constraints inherent in the user's generic actual subtype, or that the function's computed result fails to satisfy those constraints.

— It may happen that the function that is invoked is unable to obtain the storage it needs to perform the requested computation.

— It may happen that the computed result is so large, in magnitude, that it exceeds the hardware's representational capabilities—i.e., it overflows.

The first of these misfortunes can occur during any parameter association or on any function return; it is not peculiar to the functions in this package. It is a fact of life of Ada, calling for the raising of the CONSTRAINT_ERROR exception at the place of the call when it occurs during a parameter association, or at the place of the return-statement when it occurs during a function return. In the former case the function is never entered, so clearly a numerical result cannot be delivered. In the latter case the function has computed an appropriate numerical result and has attempted to deliver it but has failed, because of the range constraints that the user has imposed on the arguments and results of all the elementary functions. In marginal cases, other (slightly different) results might have been produced that do satisfy the range constraints while still satisfying the accuracy requirements, but it is not highly likely. So, for all practical purposes, it may be assumed that it is just not possible to deliver a satisfactory numerical result. By allowing CONSTRAINT_ERROR to be raised naturally when the first kind of misfortune occurs, this International Standard avoids imposing unusual design requirements on implementations. Indeed, in the case of parameter associations, there is nothing else that it could do.

The second of the three misfortunes can occur during any subprogram invocation, or during the elaboration of a subprogram's declarative part just after its invocation; it, too, is not peculiar to the functions in this package. Thus it, too, is a fact of life of Ada, calling for the raising of the STORAGE_ERROR exception at the place of the call (for all practical purposes) when it occurs for either of these reasons. Clearly, it is just not possible to deliver a numerical result, since the function either has never been entered or has not finished elaborating its declarative part. Since storage requirements for elementary function implementations (of scalar arguments) are modest, it is highly likely that the application was running near the limit of available storage at the point where the elementary function was invoked, and that it is merely an accident that the raising of STORAGE_ERROR occurred there instead of somewhere else. By allowing STORAGE_ERROR to be raised naturally when the second kind of misfortune occurs, this International Standard avoids imposing unusual design requirements on implementations. Indeed, since a handler for STORAGE_ERROR established by the function will not be entered in the cases described above, there is nothing else that it could do. (STORAGE_ERROR can also be raised by evaluation of an allocator in the sequence of statements of the function's body. Although in that case the propagation of the exception to the caller is *not* inevitable, it seemed hardly practical or appropriate for this International Standard to distinguish that case from the earlier ones.)

The last of the three misfortunes, a computed result whose magnitude is so large that it cannot be represented in the hardware, is commonly called overflow. Since the overflow threshold is a hardware property, exactly which results are "too big" cannot be predicted with certainty without reference to the hardware. The model of real arithmetic in Ada allows one to say with certainty, however, that a computed result whose absolute value is less than or equal to FLOAT_TYPE'SAFE_LARGE is always capable of being represented. Thus it is reasonable to insist that, whenever all possible results permitted by the accuracy requirements are less than or equal to FLOAT_TYPE'SAFE_LARGE in absolute value, the implementation must deliver one of them (if it does not suffer one of the earlier misfortunes). On the other hand, if any result permitted by the accuracy requirements would exceed FLOAT_TYPE'SAFE_LARGE in absolute value, then it is possible that that result is the one that the implementation might try to compute, and that it would also exceed the hardware's overflow threshold. Thus, whenever any result permitted by the accuracy requirements exceeds FLOAT_TYPE'SAFE_LARGE in absolute value, an implementation is permitted to signal overflow instead of delivering a result. That does not mean that it *must*, of course; the actual result that it computes could be some other permitted result that does not exceed FLOAT_TYPE'SAFE_LARGE in absolute value—and therefore does not exceed the hardware's overflow threshold—or it could be one that does exceed FLOAT_TYPE'SAFE_LARGE but still does not exceed the hardware's overflow threshold. By the same token, the fact that some of the permitted results do not exceed FLOAT_TYPE'SAFE_LARGE in absolute value does not oblige the implementation to deliver a result in the case that others do exceed it. This is the rationale for the variety of behaviors that an implementation is allowed to exhibit in the vicinity of the overflow threshold.

If overflow needs to be signaled, clause 12 calls for that to be done in the way that Ada mandates for its predefined operators—i.e., by raising NUMERIC_ERROR (which is changed to CONSTRAINT_ERROR by AI-00387). Implementors have a choice of ways to deal with possible overflows in the final result. On the one hand, implementors can use a predictive technique—that is, examine the arguments before using them to compute the result and raise the appropriate exception