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

Annexes

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) ≈ $\sqrt{X}$

15.1.3 Usage

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

15.1.4 Domain

X ≥ 0.0

15.1.5 Range

SQRT(X) ≥ 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) ≈ $\log_e$ X

15.2.3 Usage

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

15.2.4 Domain

X ≥ 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` · `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)` $\approx \log_{\texttt{BASE}} \texttt{X}$

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

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