INTERNATIONAL STANDARD

ISO/IEC 13814

First edition
1998-07-01

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

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

Reference number
ISO/IEC 13814:1998(E)

Contents

		Page
1	Scope	1
2	Normative references	1
3	Subprograms provided	1
4	Instantiations	2
5	Implementations	3
6	Exceptions	4
7	Arguments outside the range of safe numbers	5
8	Method of specification of functions	5
9	Branch cut and domain definitions	5
10	Range definitions	5
11	Accuracy requirements	6
12	Overflow	7
13	Underflow	8
14	Specifications of the functions	9
14.1	`SQRT` — Square root	9
14.2	`LOG` — Natural logarithm	10
14.3	`EXP` — Exponential function, complex argument	10
14.4	`EXP` — Exponential function, imaginary argument	11

14.5 "**" — Exponentiation operator, complex to complex power arguments . . . 12
14.6 "**" — Exponentiation operator, complex to real power arguments . . . 12
14.7 "**" — Exponentiation operator, real to complex power arguments . . . 13
14.8 SIN — Trigonometric sine function . . . 14
14.9 COS — Trigonometric cosine function . . . 15
14.10 TAN — Trigonometric tangent function . . . 16
14.11 COT — Trigonometric cotangent function . . . 16
14.12 ARCSIN — Inverse trigonometric sine function . . . 17
14.13 ARCCOS — Inverse trigonometric cosine function . . . 18
14.14 ARCTAN — Inverse trigonometric tangent function . . . 19
14.15 ARCCOT — Inverse trigonometric cotangent function . . . 19
14.16 SINH — Hyperbolic sine function . . . 20
14.17 COSH — Hyperbolic cosine function . . . 21
14.18 TANH — Hyperbolic tangent function . . . 22
14.19 COTH — Hyperbolic cotangent function . . . 23
14.20 ARCSINH — Inverse hyperbolic sine function . . . 23
14.21 ARCCOSH — Inverse hyperbolic cosine function . . . 24
14.22 ARCTANH — Inverse hyperbolic tangent function . . . 25
14.23 ARCCOTH — Inverse hyperbolic cotangent function . . . 25

Annexes

A Ada specification for GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS 27

B Rationale . . . 29
B.1 Introduction and motivation . . . 29
B.2 History . . . 29
B.3 Packaging considerations . . . 29
B.4 Choice of functions . . . 30
B.5 Handling of exceptions . . . 30
B.6 Matters of style . . . 32
B.7 Method of expressing accuracy requirements . . . 32

C Bibliography . . . 35

Figures

B.1 Result rectangle for a subprogram satisfying a maximum relative component error requirement . . . 33

B.2 Result rectangle for a subprogram satisfying a maximum relative box error requirement . . . 33

B.3 Result circle for a subprogram satisfying a maximum relative vector error requirement . . . 34

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 13814 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 22, *Programming languages, their environments and system software interfaces.*

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

Introduction

The generic package described here is intended to provide the basic complex 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 specification included in this International Standard is presented as a compilable Ada specification in Annex A, with explanatory text in numbered clauses in the main body of text. The explanatory text is normative, with the exception of the following items:

— notes (under the heading Notes associated with some of the functions); and

— notes (labeled as such) presented at the end of any numbered clause.

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 complex elementary functions for Ada

1 Scope

This International Standard defines the specification of a generic package of complex elementary functions called `GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS`. It does not provide the body of the package.

This International Standard specifies certain basic complex 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.

This International Standard is applicable to programming environments conforming to ISO/IEC 8652:1987 and is relevant to the revised standard ISO/IEC 8652:1995.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were 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 editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 8652, *Information technology — Programming languages — Ada.*

ISO/IEC 11430, *Information technology — Programming languages — Generic package of elementary functions for Ada.*

ISO/IEC 11729, *Information technology — Programming languages — Generic package of primitive functions for Ada.*

ISO/IEC 13813, *Information technology — Programming languages — Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types).*

3 Subprograms 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`) function 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). These are the same functions by the same names that are in the package GENERIC_ELEMENTARY_FUNCTIONS definied in ISO/IEC 11430.

Several variations are provided for the exponentiation operator and the exponential function. All functions have one or more formal parameters of type COMPLEX or IMAGINARY and return a value of type COMPLEX. COMPLEX and IMAGINARY are generic formal parameters of the generic package GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS.

4 Instantiations

This International Standard describes a generic package, GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS, which the user must instantiate to obtain a computational capability. The generic package has three required generic formal parameters and many generic formal subprograms with defaults. At instantiation, the user must specify, as the generic actual parameters to be associated with REAL, COMPLEX and IMAGINARY respectively,

a) a floating-point type,

b) a private composite type containing real and imaginary components of the type specified by a), (Within this International Standard, the real and imaginary components of the composite type are referred to simply as components.)

c) a private type of the same base type as a) that is interpreted as the pure imaginary form of b).

Types suitable for b) and c) are exported by the package obtained by instantiating, with the type specified by a) as its generic actual parameter, the generic package GENERIC_COMPLEX_TYPES defined in ISO/IEC 13813. (Instantiation of the package GENERIC_COMPLEX_TYPES is not a prerequisite for using GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS; it is merely an option that the user might find convenient to pursue.) In addition the user may specify the many subprogram generic formal parameters as generic actual parameters, or for convenience, all these subprograms are exported by the instantiation of GENERIC_COMPLEX_TYPES described above.

Depending on the implementation, the user may or may not be allowed to specify a generic actual type having a range constraint for the first generic actual parameter (see clause 5). The generic actual parameters for the subprograms can be omitted if functions having names and profiles matching those of the corresponding generic formal parameters are visible at the place of the instantiation. An instantiation of GENERIC_COMPLEX_TYPES, as described above, can optionally be used to obtain subprograms that satisfy these requirements.

In addition to the body of the generic package itself, implementers may provide (non-generic) library packages that can be used just like instantiations of the generic package in which the first generic actual parameter is a predefined floating-point type and the remainder are related to the first in appropriate ways. In particular, the name of a package serving as a replacement for an instantiation of GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS for the predefined type FLOAT shall be COMPLEX_ELEMENTARY_FUNCTIONS with REAL replaced by FLOAT in the profiles of the functions that it exports. Similarly, the names of packages serving as replacements for instantiations of GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS for the predefined types LONG_FLOAT, SHORT_FLOAT, etc., shall be LONG_COMPLEX_ELEMENTARY_FUNCTIONS, SHORT_COMPLEX_ELEMENTARY_FUNCTIONS, etc., respectively and REAL shall be systematically replaced in the profiles of the functions that they export by LONG_FLOAT, SHORT_FLOAT, etc. Implementers are responsible for ensuring that a composite type called COMPLEX containing a pair of components of types FLOAT, LONG_FLOAT, SHORT_FLOAT, etc., respectively, are available. For example, each of the non-generic library packages could be prefixed by an appropriate context clause, such as:

```
with COMPLEX_TYPES;
package COMPLEX_ELEMENTARY_FUNCTIONS is ...
```

The packages COMPLEX_TYPES, LONG_COMPLEX_TYPES, etc., described in ISO/IEC 13813, are not a prerequisite for implementation, yet are an option the implementer might find convenient to pursue.

5 Implementations

Portable implementations of GENERIC_COMPLEX_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. On the other hand, to the extent that generic packages (e.g. GENERIC_ELEMENTARY_FUNCTIONS defined in ISO/IEC 11430 and GENERIC_PRIMITIVE_FUNCTIONS defined in ISO/IEC 11729) become widely available, portable implementations of GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS should not be particularly difficult to construct. In particular, it is possible to achieve the accuracy required by this International Standard, when specified, by appropriate use of the package GENERIC_ELEMENTARY_FUNCTIONS.

An implementation is allowed to 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_COMPLEX_ELEMENTARY_FUNCTIONS is said not to conform to this International Standard in any environment in which its limits or assumptions are not satisfied, and this International Standard does not define its behavior in that environment. In effect, this convention delimits the portability of implementations.

An implementation is allowed to impose a restriction that the generic actual parameter associated with REAL 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:

a) Compilation of a unit containing an instantiation of GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS is rejected.

b) CONSTRAINT_ERROR or PROGRAM_ERROR is raised during the elaboration of an instantiation of GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS.

Conversely, if an implementation does not impose the restriction, then it must not allow such a range constraint, when included with the user's actual type, to interfere with the internal computations of the functions; that is, if the components of the argument and the 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).

Any of the restrictions discussed above may in fact be inherited from implementations of the packages GENERIC_ELEMENTARY_FUNCTIONS defined in ISO/IEC 11430, GENERIC_PRIMITIVE_FUNCTIONS defined in ISO/IEC 11729 and GENERIC_COMPLEX_TYPES defined in ISO/IEC 13813, if used. The dependence of an implementation on such inherited restrictions should be documented.

An implementation shall function properly in a tasking environment. Apart from the obvious restriction that an implementation of GENERIC_COMPLEX_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, e.g., of preserving the sign of an infinitesimal quantity that has underflowed to zero. This International Standard allows implementations of GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS to exploit that capability, when available, so as to exhibit continuity in the results of some functions as certain limits are approached. At the same time, it accommodates implementations in which that capability is unavailable. 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 available. An implementation shall exercise its choice consistently, either exploiting signed-zero behavior everywhere or nowhere in this package. The signs of zero results prescribed by this International Standard apply only to implementations that exploit signed zeros; an implementation that does not exploit signed zeros may give any sign to a zero result. In addition an implementation shall document its behavior with respect to signed zeros. In this International Standard, unless otherwise qualified, zero means any of (±0.0, ±0.0) and one means either of (1.0, ±0.0).

6 Exceptions

The ARGUMENT_ERROR exception is raised by a function in the generic package when an argument of the function violates one or more of the conditions given in the function's domain definition (see clause 9). Note that 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 package ELEMENTARY_FUNCTIONS_EXCEPTIONS defined in ISO/IEC 11430. This exception distinguishes neither between different kinds of argument errors, nor between different functions, nor between different instantiations of GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS, nor between instantiations of GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS and those of GENERIC_ELEMENTARY_FUNCTIONS.

Besides ARGUMENT_ERROR, the only exceptions allowed during a call to a function in GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS are predefined exceptions, as follows:

a) Virtually any predefined exception is possible during the evaluation of an argument of a function in GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS. For example, NUMERIC_ERROR, CONSTRAINT_ERROR or even PROGRAM_ERROR could be raised if an argument has an undefined value; and CONSTRAINT_ERROR will be raised when the value of an argument (or component of an argument) lies outside the range of the user's generic actual type REAL. 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_COMPLEX_ELEMENTARY_FUNCTIONS.

b) Also, CONSTRAINT_ERROR will be raised when a function in GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS attempts to construct a complex return value with a component outside the range of the user's generic actual type associated with REAL. The exception raised for this reason shall be propagated to the caller of the function.

c) Whenever the arguments of a function are such that a component of a result permitted by the accuracy requirements would exceed REAL'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.

d) Whenever the arguments of a function are such that the corresponding mathematical function is infinite (i.e., has an infinite component), an implementation shall raise and propagate to the caller the exception specified by Ada for signaling division by zero.

e) Once execution of a function has begun, an implementation may propagate STORAGE_ERROR to the caller of the function, but only to signal the unexpected exhaustion of storage. Similarly, once execution 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_COMPLEX_ELEMENTARY_FUNCTIONS.

No exception is allowed during a call to a function in GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS except those permitted by the foregoing rules. In particular, for arguments for which all components of results satisfying the accuracy requirements remain less than or equal to REAL'SAFE_LARGE in absolute value in both real and imaginary parts, a function shall locally handle 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_COMPLEX_ELEMENTARY_FUNCTIONS, including the execution of the optional sequence of statements in the body of the instance, are CONSTRAINT_ERROR, STORAGE_ERROR and PROGRAM_ERROR, and then only for the following reasons. The raising of CONSTRAINT_ERROR during instantiation is only allowed when the implementation imposes the restriction that the generic actual type associated with REAL shall not have a range constraint, and the user violates that restriction (it may, 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 violating some restriction or limitation of the implementation. The raising of STORAGE_ERROR during instantiation is only allowed for the purpose of signaling the exhaustion of storage.

NOTE — In ISO/IEC 8652:1995, the exception specified for signaling overflow is CONSTRAINT_ERROR.

7 Arguments outside the range of safe numbers

The ISO/IEC 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 a component of one of its arguments exceeds `REAL'SAFE_LARGE`. All of the accuracy requirements and other provisions of the following clauses are understood to be implicitly qualified by the assumption that both components of function arguments are less than or equal to `REAL'SAFE_LARGE` in absolute value.

8 Method of specification of functions

One of the functions has three overloaded forms. For each form of each function covered by this International Standard, the function is specified by its parameter and result type profile, a defining equation, the applicable branch cuts, the domain of its argument(s), its range, and the accuracy required of its implementation. The meaning of, and conventions applicable to, these specifications are described below.

The specification of each function has a heading called *Definition*. This is meant to convey the required behavior of the function. The definition is not necessarily an appropriate implementation for the function. The definition is one of the mathematically correct definitions of the function but algebraic simplifications or transformations may not be correct. Transformations may cause the sign of the result to be wrong or cause a gratuitous singularity. The definition is based on exact computation whereas an implementation must take into account the accuracy requirements.

9 Branch cut and domain definitions

The specification of each function covered by this International Standard includes headings of *Branch cut* and *Domain*. The branch cuts or slits are specified to warn the user of discontinuity in the function or one of its derivatives. Under the heading *Domain*, a characterization of the argument values is given for which the function is mathematically defined. The portion of the complex plane over which the function is defined 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, i.e., that the domain is the entire complex plane. Whenever the arguments fail to satisfy all the conditions, the implementation shall raise `ARGUMENT_ERROR`. The implementation shall not raise `ARGUMENT_ERROR` if all the domain 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).

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. Many of the functions covered by this International Standard 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.

The range of each function is shown under the heading *Range* in the specifications. The range definition is expressed by inequalities or other conditions that shall be satisfied by the components of the returned result. An implementation shall not exceed a limit of a range component when that limit is a safe number of `REAL` (like `0.0` or `1.0`). On the other hand, when a range component limit is not a safe number of `REAL` (like π), an implementation is allowed to exceed the range component limit, but it is not allowed to exceed the safe number of `REAL` next beyond the range component limit in the direction away from the interior of the range component; 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.

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 (the precise meaning of a range limit that is not a safe number of `REAL`, as an accuracy requirement, is discussed above 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 13, below. 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 separate bound on the relative error in the components of the computed value of the function, which shall hold (except as provided by the rules in clauses 12 and 13) for all arguments satisfying the conditions in the domain definition, providing the mathematical result is finite and nonzero. For some functions, the relative error bound is stated as a bound on the "relative box error" in each component, while for others it is stated as a bound on the "relative component error" in each component. For a given function f, having an exact result $f(z)$, the relative box error in the components of a computed result $F(z)$ at the argument z is defined as

relative box error in real part of result

$$= \frac{|\operatorname{re}(F(z)) - \operatorname{re}(f(z))|}{\max(|\operatorname{re}(f(z))|, |\operatorname{im}(f(z))|)}$$

relative box error in imaginary part of result

$$= \frac{|\operatorname{im}(F(z)) - \operatorname{im}(f(z))|}{\max(|\operatorname{re}(f(z))|, |\operatorname{im}(f(z))|)}$$

while the relative component error is defined as

relative component error in real part of result

$$= \frac{|\operatorname{re}(F(z)) - \operatorname{re}(f(z))|}{|\operatorname{re}(f(z))|}$$

relative component error in imaginary part of result

$$= \frac{|\operatorname{im}(F(z)) - \operatorname{im}(f(z))|}{|\operatorname{im}(f(z))|}$$

providing the denominators are finite and nonzero. The relative box error in a component of the result is not defined when the mathematical result is zero or either of its components are infinite. The relative component error in a component of the result is not defined when that component of the mathematical result is infinite or zero.

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 a component of the computed result to be zero when the relative error is undefined, i.e., when the mathematical result has a zero component, and to strengthen the accuracy requirements at special argument values. When a component of such a prescribed result is a safe number of `REAL` (like `0.0` or `1.0`), an implementation shall deliver a result having that safe number as its component. On the other hand, when a component of a prescribed result is not a safe number of `REAL` (like π or $\pi/2$), an implementation may deliver a result in which the component has any value in the surrounding safe interval. Prescribed results take precedence over maximum relative error requirements but never contravene them. The real and imaginary parts may have any combination of the two kinds of accuracy requirement.

Range definitions, under the heading *Range* in the specifications, are an additional source of accuracy requirements, as stated above 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, i.e., 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 `REAL`, that maximum will be at least `REAL'SAFE_LARGE`. For the functions defined by this International Standard, whenever the maximum error requirements permit a result with a component whose absolute value is greater than `REAL'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.

In addition, some of the functions are allowed to signal overflow for certain arguments for which neither component of the result can overflow. This freedom is granted in the following specific cases:

a) for `EXP`, when re `X` $>$ $\log_e$ `REAL'SAFE_LARGE`;

b) for `SIN`, when im `X` $>$ $\log_e$ `REAL'SAFE_LARGE` $+$ $\log_e$ `2.0`;

c) for `COS`, when im `X` $>$ $\log_e$ `REAL'SAFE_LARGE` $+$ $\log_e$ `2.0`;

d) for `SINH`, when re `X` $>$ $\log_e$ `REAL'SAFE_LARGE` $+$ $\log_e$ `2.0`;

e) for `COSH`, when re `X` $>$ $\log_e$ `REAL'SAFE_LARGE` $+$ $\log_e$ `2.0`.

Permission to signal overflow in these cases recognizes the difficulty of avoiding overflow in the computation of intermediate results and allows the same latitude as specified in the package `GENERIC_ELEMENTARY_FUNCTIONS` defined in ISO/IEC 11430 for real functions `EXP`, `SINH` and `COSH`.

An implementation shall raise the exception specified by Ada for signaling division by zero in the following specific cases:

a) `LOG(X)` when `X` is zero;

b) `LEFT ** RIGHT` (clause 14.5) when `LEFT` is zero and re `RIGHT` $<$ `0.0`;

c) `LEFT ** RIGHT` (clause 14.6) when `LEFT` is zero and `RIGHT` $<$ `0.0`;

d) `LEFT ** RIGHT` (clause 14.7) when `LEFT` $=$ `0.0` and re `RIGHT` $<$ `0.0`;

e) `COT(X)` when `X` is zero;