INTERNATIONAL STANDARD

ISO/IEC 10967-2

First edition
2001-08-15

Information technology — Language independent arithmetic —

Part 2: **Elementary numerical functions**

Technologies de l'information — Arithmétique de langage indépendant —

Partie 2: Fonctions numériques élémentaires

Reference number
ISO/IEC 10967-2:2001(E)

Contents

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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

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.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 10967 may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 10967-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces.*

ISO/IEC 10967 consists of the following parts, under the general title *Information technology — Language independent arithmetic*:

- *Part 1: Integer and floating point arithmetic*
- *Part 2: Elementary numerical functions*
- *Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions*

Additional parts will specify other arithmetic datatypes or arithmetic operations.

Annex A forms a normative part of this part of ISO/IEC 10967. Annexes B to E are for information only.

Introduction

The aims

Portability is a key issue for scientific and numerical software in today's heterogeneous computing environment. Such software may be required to run on systems ranging from personal computers to high performance pipelined vector processors and massively parallel systems, and the source code may be ported between several programming languages. Part 1 of ISO/IEC 10967 specifies the basic properties of integer and floating point types that can be relied upon in writing portable software.

Programmers writing programs that perform a significant amount of numeric processing have often not been certain how a program will perform when run under a given language processor. Programming language standards have traditionally been somewhat weak in the area of numeric processing, seldom providing an adequate specification of the properties of arithmetic datatypes, particularly floating point numbers. Often they do not even require much in the way of documentation of the actual arithmetic operations by a conforming language processor.

It is the intent of this part to help to redress these shortcomings, by setting out precise definitions of elementary numerical functions, and requirements for documentation.

It is not claimed that this part will ensure complete certainty of arithmetic behaviour in all circumstances; the complexity of numeric software and the difficulties of analysing and proving algorithms are too great for that to be attempted. Rather, this International Standard will provide a firmer basis than hitherto for attempting such analysis.

The aims for this part, part 2 of ISO/IEC 10967, are extensions of the aims for part 1: to ensure adequate accuracy for numerical computation, predictability, notification on the production of exceptional results, and compatibility with programming language standards.

The content

The content of this part is based on part 1, and extends part 1's specifications to specifications for operations approximating real elementary functions, operations often required (usually without a detailed specification) by the standards for programming languages widely used for scientific software. This part also provides specifications for conversions between the "internal" values of an arithmetic datatype, and a very close approximation in, e.g., the decimal radix. It does not cover the further transformation to decimal string format, which is usually provided by language standards. This part also includes specifications for a number of other functions deemed useful, even though they may not be stipulated by programming language standards.

The numerical functions covered by this part are computer approximations to mathematical functions of one or more real arguments. Accuracy versus performance requirements often vary with the application at hand. This is recognised by recommending that implementors support more than one library of these numerical functions. Various documentation and (program available) parameters requirements are specified to assist programmers in the selection of the library best suited to the application at hand.

The benefits

Adoption and proper use of this part can lead to the following benefits.

Language standards will be able to define their arithmetic semantics more precisely without preventing the efficient implementation of their language on a wide range of machine architectures.

Programmers of numeric software will be able to assess the portability of their programs in advance. Programmers will be able to trade off program design requirements for portability in the resulting program.

Programs will be able to determine (at run time) the crucial numeric properties of the implementation. They will be able to reject unsuitable implementations, and (possibly) to correctly characterize the accuracy of their own results. Programs will be able to detect (and possibly correct for) exceptions in arithmetic processing.

End users will find it easier to determine whether a (properly documented) application program is likely to execute satisfactorily on their platform. This can be done by comparing the documented requirements of the program against the documented properties of the platform.

Finally, end users of numeric application packages will be able to rely on the correct execution of those packages. That is, for correctly programmed algorithms, the results are reliable if and only if there is no notification.

Information technology — Language independent arithmetic —

Part 2: Elementary numerical functions

1 Scope

This part of ISO/IEC 10967 defines the properties of numerical approximations for many of the real elementary numerical functions available in standard libraries for a variety of programming languages in common use for mathematical and numerical applications.

An implementor may choose any combination of hardware and software support to meet the specifications of this part. It is the computing environment, as seen by the programmer/user, that does or does not conform to the specifications.

The term *implementation* (of this part) denotes the total computing environment pertinent to this part, including hardware, language processors, subroutine libraries, exception handling facilities, other software, and documentation.

1.1 Inclusions

The specifications of part 1 are included by reference in this part.

This part provides specifications for numerical functions for which all operand values are of integer or floating point datatypes satisfying the requirements of part 1. Boundaries for the occurrence of exceptions and the maximum error allowed are prescribed for each specified operation. Also the result produced by giving a special value operand, such as an infinity, or a NaN, is prescribed for each specified floating point operation.

This part covers most numerical functions required by the ISO/IEC standards for Ada [11], Basic [16], C [17], C++ [18], Fortran [22], ISLisp [24], Pascal [27], and PL/I [29]. In particular, specifications are provided for:

a) Some additional integer operations.

b) Some additional non-transcendental floating point operations, including maximum and minimum operations.

c) Exponentiations, logarithms, and hyperbolics.

d) Trigonometrics, both in radians and for argument-given angular unit with degrees as a special case.

This part also provides specifications for:

e) Conversions between integer and floating point datatypes (possibly with different radices) conforming to the requirements of part 1, and the conversion operations used, for example, in text input and output of integer and floating point numbers.

f) The results produced by an included floating point operation when one or more argument values are IEC 60559 special values.

g) Program-visible parameters that characterise certain aspects of the operations.

1.2 Exclusions

This part provides no specifications for

a) Numerical functions whose operands are of more than one datatype (with one exception). This part neither requires nor excludes the presence of such "mixed operand" operations.

b) An interval datatype, or the operations on such data. This part neither requires nor excludes such data or operations.

c) A fixed point datatype, or the operations on such data. This part neither requires nor excludes such data or operations.

d) A rational datatype, or the operations on such data. This part neither requires nor excludes such data or operations.

e) Complex, matrix, statistical, or symbolic operations. This part neither requires nor excludes such data or operations.

f) The properties of arithmetic datatypes that are not related to the numerical process, such as the representation of values on physical media.

g) The properties of integer and floating point datatypes that properly belong in programming language standards or other specifications. Examples include

   1) the syntax of numerals and expressions in the programming language,
   2) the syntax used for parsed (input) or generated (output) character string forms for numerals by any specific programming language or library,
   3) the precedence of operators in the programming language,
   4) the presence or absence of automatic datatype coercions,
   5) the rules for assignment, parameter passing, and returning value,
   6) the consequences of applying an operation to values of improper datatype, or to uninitialised data.

Furthermore, this part does not provide specifications for how the operations should be implemented or which algorithms are to be used for the various operations.

2 Conformity

It is expected that the provisions of this part of ISO/IEC 10967 will be incorporated by reference and further defined in other International Standards; specifically in programming language standards and in binding standards.

A binding standard specifies the correspondence between one or more of the parameters and operations specified in this part and the concrete language syntax of some programming language. More generally, a binding standard specifies the correspondence between certain parameters and operations and the elements of some arbitrary computing entity. A language standard that explicitly provides such binding information can serve as a binding standard.

When a binding standard for a language exists, an implementation shall be said to conform to this part if and only if it conforms to the binding standard. In case of conflict between a binding standard and this part, the specification of the binding standard takes precedence.

When a binding standard covers only a subset of the operations specified in this part, an implementation remains free to conform to this part with respect to other operations, independently of that binding standard.

When no binding standard for a language and some operations specified in this part exists, an implementation conforms to this part if and only if it provides one or more operations that together satisfy all the requirements of clauses 5 through 8 that are relevant to those operations. The implementation shall then document the binding.

Conformity to this part is always with respect to a specified set of datatypes and operations. Conformity to this part implies conformity to part 1 for the integer and floating point datatypes used.

An implementation is free to provide operations that do not conform to this part, or that are beyond the scope of this part. The implementation shall not claim or imply conformity to this part with respect to such operations.

An implementation is permitted to have modes of operation that do not conform to this part. A conforming implementation shall specify how to select the modes of operation that ensure conformity. However, a mode of operation that conforms to this part should be the default mode of operation.

NOTES

1 Language bindings are essential. Clause 8 requires an implementation to supply a binding if no binding standard exists. See annex C for suggested language bindings.

2 A complete binding for this part will include (explicitly or by reference) a binding for part 1 as well, which in turn may include (explicitly or by reference) a binding for IEC 60559 as well.

3 This part does not require a particular set of operations to be provided. It is not possible to conform to this part without specifying to which datatypes and set of operations (and modes of operation) conformity is claimed.

3 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 10967. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 10967 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems.*

ISO/IEC 10967-1:1994, *Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic.*

NOTE – See also annex E.

4 Symbols and definitions

4.1 Symbols

4.1.1 Sets and intervals

In this part, $\mathcal{Z}$ denotes the set of mathematical integers, $\mathcal{R}$ denotes the set of classical real numbers, and $\mathcal{C}$ denotes the set of complex numbers over $\mathcal{R}$. Note that $\mathcal{Z} \subset \mathcal{R} \subset \mathcal{C}$.

The conventional notation for set definition and manipulation is used.

In this part, the following notation for intervals is used

$[x, z]$ designates the interval $\{y \in \mathcal{R} \mid x \leqslant y \leqslant z\}$,
$]x, z]$ designates the interval $\{y \in \mathcal{R} \mid x < y \leqslant z\}$,
$[x, z[$ designates the interval $\{y \in \mathcal{R} \mid x \leqslant y < z\}$, and
$]x, z[$ designates the interval $\{y \in \mathcal{R} \mid x < y < z\}$.

NOTE – The notation using a round bracket for an open end of an interval is not used, for the risk of confusion with the notation for pairs.

4.1.2 Operators and relations

All prefix and infix operators have their conventional (exact) mathematical meaning. In particular this part uses

$\Rightarrow$ and $\Leftrightarrow$ for logical implication and equivalence
$+$, $-$, $/$, $|x|$, $\lfloor x \rfloor$, $\lceil x \rceil$, and round(x) on reals
$\cdot$ for multiplication on reals
$<$, $\leqslant$, $\geqslant$, and $>$ between reals
$=$ and $\neq$ between real as well as special values
max on non-empty upwardly closed sets of reals
min on non-empty downwardly closed sets of reals
$\cup$, $\cap$, $\times$, $\in$, $\notin$, $\subset$, $\subseteq$, $\nsubseteq$, $\neq$, and $=$ with sets
$\times$ for the Cartesian product of sets
$\rightarrow$ for a mapping between sets
$\mid$ for the divides relation between integers

For $x \in \mathcal{R}$, the notation $\lfloor x \rfloor$ designates the largest integer not greater than x:

$\lfloor x \rfloor \in \mathcal{Z}$ and $x - 1 < \lfloor x \rfloor \leqslant x$

the notation $\lceil x \rceil$ designates the smallest integer not less than x:

$\lceil x \rceil \in \mathcal{Z}$ and $x \leqslant \lceil x \rceil < x + 1$

and the notation round(x) designates the integer closest to x:

$$\text{round}(x) \in \mathcal{Z} \quad \text{and} \quad x - 0.5 \leqslant \text{round}(x) \leqslant x + 0.5$$

where in case x is exactly half-way between two integers, the even integer is the result.

The *divides* relation ($|$) on integers tests whether an integer i divides an integer j exactly:

$$i|j \;\Leftrightarrow\; (i \neq 0 \text{ and } i \cdot n = j \text{ for some } n \in \mathcal{Z})$$

NOTE – $i|j$ is true exactly when j/i is defined and $j/i \in \mathcal{Z}$).

4.1.3 Mathematical functions

This part specifies properties for a number of operations numerically approximating some of the elementary functions. The following ideal mathematical functions are defined in chapter 4 of the *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* [47] (e is the Napierian base)

e^x, x^y, $\sqrt{x}$, ln, $\log_b$,
sin, cos, tan, cot, sec, csc, arcsin, arccos, arctan, arccot, arcsec, arccsc,
sinh, cosh, tanh, coth, sech, csch, arcsinh, arccosh, arctanh, arccoth, arcsech, arccsch.

Many of the inverses above are multi-valued. The selection of which value to return, the principal value, so as to make the inverses into functions, is done in the conventional way. E.g., $\sqrt{x} \in [0, \infty[$ when $x \in [0, \infty[$. The only one over which there is some difference of conventions is the arccot function. Conventions there vary for negative arguments; either a negative value (giving a sign symmetric function), or a positive return value (giving a function that is continuous over zero). In this part, arccot refers to the sign symmetric inverse function (with a branch cut at 0), and arccotc refers to the continuous inverse function.

$\text{arccosh}(x) \geqslant 0$, $\text{arcsech}(x) \geqslant 0$,
$\arcsin(x) \in [-\pi/2, \pi/2]$, $\arccos(x) \in [0, \pi]$, $\arctan(x) \in]-\pi/2, \pi/2[$,
$\text{arccot}(x) \in]-\pi/2, \pi/2]$, $\text{arccotc}(x) \in]0, \pi[$, $\text{arcsec}(x) \in [0, \pi]$, $\text{arccsc}(x) \in [-\pi/2, \pi/2]$.

NOTE – $e = 2.71828....$ e is not in any floating point datatype conforming to part 1, unless added as a special value, which is usually not done.

4.1.4 Exceptional values

The exceptional value **underflow** is used in this part as it is in part 1.

Three new exceptional values, **overflow**, **invalid**, and **infinitary**, are introduced in this part replacing three other exceptional values used in part 1. **invalid** and **infinitary** are in this part used instead of the **undefined** of part 1. **overflow** is used instead of the **integer_overflow** and **floating_overflow** of part 1. Bindings may still distinguish between **integer_overflow** and **floating_overflow**.

One new exceptional value, **absolute_precision_underflow**, is introduced in this part with no correspondence in part 1. The exceptional value **absolute_precision_underflow** is used when the given floating point angle value argument is so big that even a highly accurate result from a trigonometric operation is questionable, due to the fact that the density of floating point values has decreased significantly at these big angle values.

For the exceptional values, a continuation value may be given in parenthesis after the exceptional value.

4.1.5 Datatypes

The datatype **Boolean** consists of the two values **true** and **false**.

NOTE 1 – Mathematical relations are true or false (or undefined, if an operand is undefined). In contrast, **true** and **false** are values in **Boolean**.

For pairs, define:

$fst((x, y)) = x$
$snd((x, y)) = y$

Square brackets are used to write finite sequences of values. [] is the sequence containing no values. $[s]$, is the sequence of one value, s. $[s_1, s_2]$, is the sequence of two values, s_1 and then s_2, etc. The colon operator is used to prepend a value to a sequence: $x : [x_1, ..., x_n] = [x, x_1, ..., x_n]$. $[S]$, where S is a set, denotes the set of finite sequences, where each value in a sequence is in S.

NOTE 2 – It is always clear from context, in the text of this part, if $[X]$ is a sequence of one element, or the set of sequences with values from X. It is also clear from context if $[x_1, x_2]$ is a sequence of two values or an interval.

Integer datatypes and floating point datatypes are defined in part 1. Let I be the non-special value set for an integer datatype conforming to part 1. Let F be the non-special value set for a floating point datatype conforming to part 1. Floating point datatypes that conform to part 1 shall, for use with this part, have a value for the parameter p_F such that $p_F \geqslant 2 \cdot \max\{1, \lceil \log_{r_F}(2 \cdot \pi)\rceil\}$, and have a value for the parameter $emin_F$ such that $emin_F \leqslant -p_F - 1$.

NOTES

3 This implies that $fminN_F < 0.5 \cdot epsilon_F / r_F$ in this part, rather than just $fminN_F \leqslant epsilon_F$.

4 These extra requirements, which do not limit the use of any existing floating point datatype, are made 1) so that angles in radians are not too degenerate within the first two cycles, plus and minus, when represented in F, and 2) in order to be able to avoid the underflow notification in specifications for the $expm1_F$ and $ln1p_F$ operations.

5 F should also be such that $p_F \geqslant 2 + \lceil \log_{r_F}(1000)\rceil$, to allow for a not too coarse angle resolution anywhere in the interval $[-big\_angle\_r_F, big\_angle\_r_F]$. See clause 5.3.8.

The following symbols represent special values defined in IEC 60559 and used in this part:

−0, **+∞**, **−∞**, **qNaN**, and **sNaN**.

These values are not part of the set I or the set F, but if iec_559_F has the value **true**, these values are included in the floating point datatype corresponding to F.

NOTE 6 – This part uses the above five special values for compatibility with IEC 60559. In particular, the symbol **−0** (in bold) is not the application of (mathematical) unary − to the value 0, and is a value logically distinct from 0.

The specifications cover the results to be returned by an operation if given one or more of the IEC 60559 special values **−0**, **+∞**, **−∞**, or NaNs as input values. These specifications apply only to systems which provide and support these special values. If an implementation is not capable of representing a **−0** result or continuation value, the actual result or continuation value shall be 0. If an implementation is not capable of representing a prescribed result or continuation value of the IEC 60559 special values $+\infty$, $-\infty$, or **qNaN**, the actual result or continuation value is binding or implementation defined.

The following symbols used in this part are defined in part 1:

Integer parameters:
$bounded_I$, $maxint_I$, and $minint_I$.
Integer helper function:
$wrap_I$.
Integer operations:
neg_I, add_I, sub_I, and mul_I.
Floating point parameters:
r_F, p_F, $emin_F$, $emax_F$, $denorm_F$, and iec_559_F.
Derived floating point constants:
$fmax_F$, $fmin_F$, $fminN_F$, $fminD_F$, and $epsilon_F$.
Floating point rounding constant:
rnd_error_F.

Floating point value sets related to F:
F^*, F_D, and F_N.
Floating point helper functions:
e_F, $result_F$, and rnd_F.
Floating point operations:
neg_F, add_F, sub_F, mul_F, div_F, and ulp_F.

4.2 Definitions of terms

For the purposes of this part, the following definitions apply:

accuracy: The closeness between the true mathematical result and a computed result.

arithmetic datatype: A datatype whose non-special values are members of $\mathcal{Z}$, $\mathcal{R}$, or $\mathcal{C}$.

NOTE 1 – This part specifies requirements for integer and floating point datatypes. Complex numbers are not covered by this part, but will be included in a subsequent part of ISO/IEC 10967 [3].

continuation value: A computational value used as the result of an arithmetic operation when an exception occurs. Continuation values are intended to be used in subsequent arithmetic processing. A continuation value can be a (in the datatype representable) value in $\mathcal{R}$ or an IEC 60559 special value. (Contrast with *exceptional value*. See clause 6.1 of part 1.)

denormalisation loss: A larger than normal rounding error caused by the fact that subnormal values have less than full precision. (See clause 5.2 of part 1 for a full definition.)

error: (1) The difference between a computed value and the correct value. (Used in phrases like "rounding error" or "error bound".)

(2) A synonym for *exception* in phrases like "error message" or "error output". Error and exception are not synonyms in any other context.

exception: The inability of an operation to return a suitable finite numeric result from finite arguments. This might arise because no such finite result exists mathematically (**infinitary**

(e.g. at a pole), **invalid** (e.g. when the true result is in $\mathcal{C}$ but not in $\mathcal{R}$), or because the mathematical result cannot, or might not, be representable with sufficient accuracy (**underflow**, **overflow**) or viability (**absolute_precision_underflow**).

NOTE 2 – The term exception is here not used to designate certain methods of handling notifications that fall under the category 'change of control flow'. Such methods of notification handling will be referred to as "[programming language name] exception", when referred to, particularly in annex C.

exceptional value: A non-numeric value produced by an arithmetic operation to indicate the occurrence of an exception. Exceptional values are not used in subsequent arithmetic processing. (See clause 5 of part 1.)

NOTES

3 Exceptional values are used as part of the defining formalism only. With respect to this part, they do not represent values of any of the datatypes described. There is no requirement that they be represented or stored in the computing system.

4 Exceptional values are not to be confused with the NaNs and infinities defined in IEC 60559. Contrast this definition with that of *continuation value* above.

helper function: A function used solely to aid in the expression of a requirement. Helper functions are not visible to the programmer, and are not required to be part of an implementation. However, some implementation defined helper functions are required to be documented.

implementation (of this part)**:** The total arithmetic environment presented to a programmer, including hardware, language processors, exception handling facilities, subroutine libraries, other software, and documentation pertinent to this part.

literal: A syntactic entity, that does not have any proper sub-entity that is an expression, denoting a constant value.

monotonic approximation: An approximation helper function $h : ... \times S \times ... \to \mathcal{R}$, where the other arguments are kept constant, and where $S \subseteq \mathcal{R}$, is a monotonic approximation of a predetermined mathematical function $f : \mathcal{R} \to \mathcal{R}$ if, for every $a \in S$ and $b \in S$, where $a < b$,

a) f is monotonic non-decreasing on $[a, b]$ implies $h(..., a, ...) \leqslant h(..., b, ...)$,

b) f is monotonic non-increasing on $[a, b]$ implies $h(..., a, ...) \geqslant h(..., b, ...)$.

monotonic non-decreasing: A function $f : \mathcal{R} \to \mathcal{R}$ is monotonic non-decreasing on a real interval $[a, b]$ if for every x and y such that $a \leqslant x \leqslant y \leqslant b$, $f(x)$ and $f(y)$ are well-defined and $f(x) \leqslant f(y)$.

monotonic non-increasing: A function $f : \mathcal{R} \to \mathcal{R}$ is monotonic non-increasing on a real interval $[a, b]$ if for every x and y such that $a \leqslant x \leqslant y \leqslant b$, $f(x)$ and $f(y)$ are well-defined and $f(x) \geqslant f(y)$.

normalised: The non-zero values of a floating point type F that provide the full precision allowed by that type. (See F_N in clause 5.2 of part 1 for a full definition.)

notification: The process by which a program (or that program's end user) is informed that an arithmetic exception has occurred. For example, dividing 2 by 0 results in a notification. (See clause 6 of part 1 for details.)

numeral: A numeric literal. It may denote a value in $\mathcal{Z}$ or $\mathcal{R}$, $\mathbf{-0}$, an infinity, or a NaN.

numerical function: A computer routine or other mechanism for the approximate evaluation of a mathematical function.

operation: A function directly available to the programmer, as opposed to helper functions or theoretical mathematical functions.

pole: A mathematical function f has a pole at x_0 if x_0 is finite, f is defined, finite, monotone, and continuous in at least one side of the neighbourhood of x_0, and $\lim_{x \to x_0} f(x)$ is infinite.

precision: The number of digits in the fraction of a floating point number. (See clause 5.2 of part 1.)

rounding: The act of computing a representable final result for an operation that is close to the exact (but unrepresentable) result for that operation. Note that a suitable representable result may not exist (see clause 5.2 of part 1).

rounding function: Any function $rnd : \mathcal{R} \to X$ (where X is a given discrete and unlimited subset of $\mathcal{R}$) that maps each element of X to itself, and is monotonic non-decreasing. Formally, if x and y are in $\mathcal{R}$,

$$x \in X \Rightarrow rnd(x) = x$$
$$x < y \Rightarrow rnd(x) \leqslant rnd(y)$$

Thus, if u is between two adjacent values in X, $rnd(u)$ selects one of those adjacent values.

round to nearest: The property of a rounding function rnd that when $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects the one nearest u. If the adjacent values are equidistant from u, either may be chosen deterministically, but so that $rnd(-u) = -rnd(u)$.

round toward minus infinity: The property of a rounding function rnd that when $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects the one less than u.

round toward plus infinity: The property of a rounding function rnd that when $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects the one greater than u.

shall: A verbal form used to indicate requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted. (Quoted from the directives [1].)

should: A verbal form used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that (in the negative form) a certain possibility is deprecated but not prohibited. (Quoted from the directives [1].)

signature (of a function or operation)**:** A summary of information about an operation or function. A signature includes the function or operation name; a subset of allowed argument values to the operation; and a superset of results from the function or operation (including exceptional values if any), if the argument is in the subset of argument values given in the signature. Approximation helper functions may be undefined for some argument values.

The signature $add_I : I \times I \to I \cup \{\textbf{overflow}\}$ states that the operation named add_I shall accept any pair of values in I as input, and when given such input shall return either a single value in I as its output or the exceptional value **overflow** possibly accompanied by a continuation value.

A signature for an operation or function does not forbid the operation from accepting a wider range of arguments, nor does it guarantee that every value in the result range will actually be returned for some argument(s). An operation given an argument outside the stipulated argument domain may produce a result outside the stipulated result range.

subnormal: The non-zero values of a floating point type F that provide less than the full precision allowed by that type. (See F_D in clause 5.2 of part 1 for a full definition. In part 1 and IEC 60559 this concept is called denormal.)

ulp: The value of one "unit in the last place" of a floating point number. This value depends on the exponent, the radix, and the precision used in representing the number. Thus, the ulp of a normalised value x (in F), with exponent t, precision p_F, and radix r_F, is $r_F^{t-p_F}$, and the ulp of a subnormal or zero value is $fminD_F$. (See clause 5.2 of part 1.)

5 Specifications for integer and floating point operations

This clause specifies a number of helper functions and operations for integer and floating point datatypes. Each operation is given a signature and is further specified by a number of cases. These cases may refer to other operations (specified in this part or in part 1), to mathematical functions, and to helper functions (specified in this part or in part 1). They also use special abstract values (**−∞**, **+∞**, **−0**, **qNaN**, **sNaN**). For each datatype, two of these abstract values may represent several actual values each: **qNaN** and **sNaN**. Finally, the specifications may refer to exceptional values.

The signatures in the specifications in this clause specify only all non-special values as input values, and indicate as output values a superset of all non-special, special, and exceptional values that may result from these (non-special) input values. Exceptional and special values that can never result from non-special input values are not included in the signatures given. Also, signatures that, for example, include IEC 60559 special values as arguments are not given in the specifications below. This does not exclude such signatures from being valid for these operations.

5.1 Basic integer operations

Clause 5.1 of part 1 specifies integer datatypes and a number of operations on values of an integer datatype. In this clause some additional operations on values of an integer datatype are specified.

I is the set of non-special values, $I \subseteq \mathcal{Z}$, for an integer datatype conforming to part 1. Integer datatypes conforming to part 1 often do not contain any NaN or infinity values, even though they may do so. Therefore this clause has no specifications for such values as arguments or results other than as continuation values.

NOTE – For some integer operations, **infinitary** notifications may occur. For **infinitary** notifications, an infinitary continuation value is recommended. For bounded integer datatypes, $maxint_I$ or $minint_I$ may be used as replacement continuation values as appropriate, if infinitary values are not available in the datatype. For unbounded integer datatypes, however, no $maxint_I$ and $minint_I$ in I are defined, and infinitary values should be used.

5.1.1 The integer *result* and *wrap* helper functions

The $result_I$ helper function:

$result_I : \mathcal{Z} \rightarrow I \cup \{\mathbf{overflow}\}$

$$\begin{array}{lll} result_I(x) & = x & \text{if } x \in I \\ & = \textbf{overflow} & \text{if } x \in \mathcal{Z} \text{ and } x \notin I \end{array}$$

The $wrap_I$ helper function:

$wrap_I : \mathcal{Z} \to I$

$$\begin{array}{lll} wrap_I(x) & = x & \text{if } x \in I \\ & = x - (n \cdot (maxint_I - minint_I + 1)) & \\ & & \text{if } x \in \mathcal{Z} \text{ and } x \notin I \end{array}$$

where $n \in \mathcal{Z}$ is chosen such that the result is in I.

NOTES

1 $n = \lfloor (x - minint_I)/(maxint_I - minint_I + 1) \rfloor$ if $x \in \mathcal{Z}$ and $bounded_I = \textbf{true}$; or equivalently $n = \lceil (x - maxint_I)/(maxint_I - minint_I + 1) \rceil$ if $x \in \mathcal{Z}$ and $bounded_I = \textbf{true}$.

2 For some wrapping basic arithmetic operations this n is computed by the '*_ov*' operations in clause 5.1.9.

3 The $wrap_I$ helper function is also used in part 1.

5.1.2 Integer maximum and minimum

$max_I : I \times I \to I$

$$\begin{array}{lll} max_I(x, y) & = \max\{x, y\} & \text{if } x, y \in I \end{array}$$

$min_I : I \times I \to I$

$$\begin{array}{lll} min_I(x, y) & = \min\{x, y\} & \text{if } x, y \in I \end{array}$$

$max\_seq_I : [I] \to I \cup \{\textbf{infinitary}\}$

$$\begin{array}{lll} max\_seq_I([x_1, ..., x_n]) & & \\ & = \textbf{infinitary}(-\infty) & \text{if } n = 0 \\ & = \max\{x_1, ..., x_n\} & \text{if } n \geqslant 1 \text{ and } \{x_1, ..., x_n\} \subseteq I \end{array}$$

$min\_seq_I : [I] \to I \cup \{\textbf{infinitary}\}$

$$\begin{array}{lll} min\_seq_I([x_1, ..., x_n]) & & \\ & = \textbf{infinitary}(+\infty) & \text{if } n = 0 \\ & = \min\{x_1, ..., x_n\} & \text{if } n \geqslant 1 \text{ and } \{x_1, ..., x_n\} \subseteq I \end{array}$$

5.1.3 Integer diminish

$dim_I : I \times I \to I \cup \{\textbf{overflow}\}$

$$\begin{array}{lll} dim_I(x, y) & = result_I(\max\{0, x - y\}) & \text{if } x, y \in I \end{array}$$

NOTE – dim_I cannot be implemented as $max_I(0, sub_I(x, y))$ for bounded integer types, since this latter expression has other overflow properties.

5.1.4 Integer power and arithmetic shift

$power_I : I \times I \rightarrow I \cup \{\mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\}$

$$\begin{aligned}
power_I(x, y) &= result_I(x^y) && \text{if } x, y \in I \text{ and } (y > 0 \text{ or } |x| = 1)\\
&= 1 && \text{if } x \in I \text{ and } x \neq 0 \text{ and } y = 0\\
&= \mathbf{invalid}(1) && \text{if } x = 0 \text{ and } y = 0\\
&= \mathbf{infinitary}(+\infty) && \text{if } x = 0 \text{ and } y \in I \text{ and } y < 0\\
&= \mathbf{invalid}(0) && \text{if } x, y \in I \text{ and } x \notin \{-1, 0, 1\} \text{ and } y < 0
\end{aligned}$$

$shift2_I : I \times I \rightarrow I \cup \{\mathbf{overflow}\}$

$$shift2_I(x, y) = result_I(\lfloor x \cdot 2^y \rfloor) \quad \text{if } x, y \in I$$

$shift10_I : I \times I \rightarrow I \cup \{\mathbf{overflow}\}$

$$shift10_I(x, y) = result_I(\lfloor x \cdot 10^y \rfloor) \quad \text{if } x, y \in I$$

5.1.5 Integer square root

$sqrt_I : I \rightarrow I \cup \{\mathbf{invalid}\}$

$$\begin{aligned}
sqrt_I(x) &= \lfloor \sqrt{x} \rfloor && \text{if } x \in I \text{ and } x \geqslant 0\\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \in I \text{ and } x < 0
\end{aligned}$$

5.1.6 Divisibility tests

$divides_I : I \times I \rightarrow \mathbf{Boolean}$

$$\begin{aligned}
divides_I(x, y) &= \mathbf{true} && \text{if } x, y \in I \text{ and } x|y\\
&= \mathbf{false} && \text{if } x, y \in I \text{ and not } x|y
\end{aligned}$$

NOTES

1 $divides_I(0, 0) = \mathbf{false}$, since 0 does not divide anything, not even 0.

2 $divides_I$ cannot be implemented as, e.g., $eq_I(0, mod_I(y, x))$, since the remainder functions give notifications for a zero second argument.

$even_I : I \rightarrow \mathbf{Boolean}$

$$\begin{aligned}
even_I(x) &= \mathbf{true} && \text{if } x \in I \text{ and } 2|x\\
&= \mathbf{false} && \text{if } x \in I \text{ and not } 2|x
\end{aligned}$$

$odd_I : I \rightarrow \mathbf{Boolean}$

$$\begin{aligned}
odd_I(x) &= \mathbf{true} && \text{if } x \in I \text{ and not } 2|x\\
&= \mathbf{false} && \text{if } x \in I \text{ and } 2|x
\end{aligned}$$

5.1.7 Integer division (with floor, round, or ceiling) and remainder

$quot_I : I \times I \rightarrow I \cup \{\textbf{overflow}, \textbf{infinitary}, \textbf{invalid}\}$

$$\begin{aligned}
quot_I(x, y) &= result_I(\lfloor x/y \rfloor) && \text{if } x, y \in I \text{ and } y \neq 0\\
&= \textbf{infinitary}(\mathbf{+\infty}) && \text{if } x \in I \text{ and } x > 0 \text{ and } y = 0\\
&= \textbf{invalid}(\textbf{qNaN}) && \text{if } x = 0 \text{ and } y = 0\\
&= \textbf{infinitary}(\mathbf{-\infty}) && \text{if } x \in I \text{ and } x < 0 \text{ and } y = 0
\end{aligned}$$

NOTE – $quot_I(minint_I, -1)$, for a bounded signed integer datatype where $minint_I = -maxint_I - 1$, is the only case where this operation will overflow.

$mod_I : I \times I \rightarrow I \cup \{\textbf{invalid}\}$

$$\begin{aligned}
mod_I(x, y) &= x - (\lfloor x/y \rfloor \cdot y) && \text{if } x, y \in I \text{ and } y \neq 0\\
&= \textbf{invalid}(\textbf{qNaN}) && \text{if } x \in I \text{ and } y = 0
\end{aligned}$$

$ratio_I : I \times I \rightarrow I \cup \{\textbf{overflow}, \textbf{infinitary}, \textbf{invalid}\}$

$$\begin{aligned}
ratio_I(x, y) &= result_I(\text{round}(x/y)) && \text{if } x, y \in I \text{ and } y \neq 0\\
&= \textbf{infinitary}(\mathbf{+\infty}) && \text{if } x \in I \text{ and } x > 0 \text{ and } y = 0\\
&= \textbf{invalid}(\textbf{qNaN}) && \text{if } x = 0 \text{ and } y = 0\\
&= \textbf{infinitary}(\mathbf{-\infty}) && \text{if } x \in I \text{ and } x < 0 \text{ and } y = 0
\end{aligned}$$

$residue_I : I \times I \rightarrow I \cup \{\textbf{overflow}, \textbf{invalid}\}$

$$\begin{aligned}
residue_I(x, y) &= result_I(x - (\text{round}(x/y) \cdot y))\\
& && \text{if } x, y \in I \text{ and } y \neq 0\\
&= \textbf{invalid}(\textbf{qNaN}) && \text{if } x \in I \text{ and } y = 0
\end{aligned}$$

$group_I : I \times I \rightarrow I \cup \{\textbf{overflow}, \textbf{infinitary}, \textbf{invalid}\}$

$$\begin{aligned}
group_I(x, y) &= result_I(\lceil x/y \rceil) && \text{if } x, y \in I \text{ and } y \neq 0\\
&= \textbf{infinitary}(\mathbf{+\infty}) && \text{if } x \in I \text{ and } x > 0 \text{ and } y = 0\\
&= \textbf{invalid}(\textbf{qNaN}) && \text{if } x = 0 \text{ and } y = 0\\
&= \textbf{infinitary}(\mathbf{-\infty}) && \text{if } x \in I \text{ and } x < 0 \text{ and } y = 0
\end{aligned}$$

$pad_I : I \times I \rightarrow I \cup \{\textbf{invalid}\}$

$$\begin{aligned}
pad_I(x, y) &= (\lceil x/y \rceil \cdot y) - x && \text{if } x, y \in I \text{ and } y \neq 0\\
&= \textbf{invalid}(\textbf{qNaN}) && \text{if } x \in I \text{ and } y = 0
\end{aligned}$$

5.1.8 Greatest common divisor and least common positive multiple

$gcd_I : I \times I \rightarrow I \cup \{\textbf{overflow}, \textbf{infinitary}\}$

$$\begin{aligned}
gcd_I(x, y) &= result_I(\max\{v \in \mathcal{Z} \mid v|x \text{ and } v|y\})\\
& && \text{if } x, y \in I \text{ and } (x \neq 0 \text{ or } y \neq 0)\\
&= \textbf{infinitary}(\mathbf{+\infty}) && \text{if } x = 0 \text{ and } y = 0
\end{aligned}$$

$lcm_I : I \times I \rightarrow I \cup \{\textbf{overflow}\}$

$$\begin{aligned} lcm_I(x,y) &= result_I(\min\{v \in \mathcal{Z} \mid x|v \text{ and } y|v \text{ and } v > 0\}) \\ &\qquad \text{if } x, y \in I \text{ and } x \neq 0 \text{ and } y \neq 0 \\ &= 0 \qquad \text{if } x, y \in I \text{ and } (x = 0 \text{ or } y = 0) \end{aligned}$$

$gcd\_seq_I : [I] \rightarrow I \cup \{\textbf{overflow}, \textbf{infinitary}\}$

$$\begin{aligned} &gcd\_seq_I([x_1, ..., x_n]) \\ &\quad = result_I(\max\{v \in \mathcal{Z} \mid v|x_i \text{ for all } i \in \{1, ..., n\}\}) \\ &\qquad \text{if } \{x_1, ..., x_n\} \subseteq I \text{ and } \{x_1, ..., x_n\} \not\subseteq \{0\} \\ &\quad = \textbf{infinitary}(+\infty) \qquad \text{if } \{x_1, ..., x_n\} \subseteq \{0\} \end{aligned}$$

$lcm\_seq_I : [I] \rightarrow I \cup \{\textbf{overflow}\}$

$$\begin{aligned} &lcm\_seq_I([x_1, ..., x_n]) \\ &\quad = result_I(\min\{v \in \mathcal{Z} \mid x_i|v \text{ for all } i \in \{1, ..., n\} \text{ and } v > 0\}) \\ &\qquad \text{if } \{x_1, ..., x_n\} \subseteq I \text{ and } 0 \notin \{x_1, ..., x_n\} \\ &\quad = 0 \qquad \text{if } \{x_1, ..., x_n\} \subseteq I \text{ and } 0 \in \{x_1, ..., x_n\} \end{aligned}$$

NOTE – These specifications imply: $gcd\_seq_I([]) = \textbf{infinitary}(+\infty)$ and $lcm\_seq_I([]) = 1$.

5.1.9 Support operations for extended integer range

These operations can be used to implement extended range integer datatypes, including unbounded integer datatypes.

$add\_wrap_I : I \times I \rightarrow I$

$$add\_wrap_I(x,y) = wrap_I(x + y) \qquad \text{if } x, y \in I$$

$add\_ov_I : I \times I \rightarrow \{-1, 0, 1\}$

$$\begin{aligned} add\_ov_I(x,y) &= ((x + y) - add\_wrap_I(x,y))/(maxint_I - minint_I + 1) \\ &\qquad \text{if } x, y \in I \text{ and } bounded_I = \textbf{true} \\ &= 0 \qquad \text{if } x, y \in I \text{ and } bounded_I = \textbf{false} \end{aligned}$$

$sub\_wrap_I : I \times I \rightarrow I$

$$sub\_wrap_I(x,y) = wrap_I(x - y) \qquad \text{if } x, y \in I$$

$sub\_ov_I : I \times I \rightarrow \{-1, 0, 1\}$

$$\begin{aligned} sub\_ov_I(x,y) &= ((x - y) - sub\_wrap_I(x,y))/(maxint_I - minint_I + 1) \\ &\qquad \text{if } x, y \in I \text{ and } bounded_I = \textbf{true} \\ &= 0 \qquad \text{if } x, y \in I \text{ and } bounded_I = \textbf{false} \end{aligned}$$

$mul\_wrap_I : I \times I \rightarrow I$

$$mul\_wrap_I(x,y) = wrap_I(x \cdot y) \qquad \text{if } x, y \in I$$

$$\begin{aligned}
&mul\_ov_I : I \times I \to I\\
&mul\_ov_I(x, y) &&= ((x \cdot y) - mul\_wrap_I(x, y))/(maxint_I - minint_I + 1)\\
& && \qquad\qquad\qquad\qquad\qquad \text{if } x, y \in I \text{ and } bounded_I = \textbf{true}\\
& &&= 0 \qquad\qquad\qquad\qquad\quad\ \ \text{if } x, y \in I \text{ and } bounded_I = \textbf{false}
\end{aligned}$$

NOTE – The add_ov_I and sub_ov_I will only return -1 (for negative overflow), 0 (no overflow), and 1 (for positive overflow).

5.2 Basic floating point operations

Clause 5.2 of part 1 specifies floating point datatypes and a number of operations on values of a floating point datatype. In this clause some additional operations on values of a floating point datatype are specified.

NOTE – Further operations on values of a floating point datatype, for elementary floating point numerical functions, are specified in clause 5.3.

F is the non-special value set, $F \subset \mathcal{R}$, for a floating point datatype conforming to part 1. Floating point datatypes conforming to part 1 often do contain $\mathbf{-0}$, infinity, and NaN values. Therefore, in this clause there are specifications for such values as arguments.

5.2.1 The rounding and floating point *result* helper functions

Floating point rounding helper functions (F^* is defined in part 1):

The floating point helper function

$down_F : \mathcal{R} \to F^*$

is the rounding function that rounds towards negative infinity. The floating point helper function

$up_F : \mathcal{R} \to F^*$

is the rounding function that rounds towards positive infinity. The floating point helper function

$nearest_F : \mathcal{R} \to F^*$

is the rounding function that rounds to nearest. $nearest_F$ is partially implementation defined: the handling of ties is implementation defined, but must be sign symmetric. If $iec\_559_F = \textbf{true}$, the semantics of $nearest_F$ is completely defined by IEC 60559: in this case ties are rounded so that the result has an even last digit.

$result_F$ is a helper function that is partially implementation defined. $result_F$ has a signature:

$result_F : \mathcal{R} \times (\mathcal{R} \to F^*) \to F \cup \{\textbf{underflow}, \textbf{overflow}\}$

For the overflow cases it is defined as:

$$\begin{aligned}
&result_F(x, nearest_F) &&= \textbf{overflow}(+\boldsymbol{\infty}) && \text{if } x \in \mathcal{R} \text{ and } nearest_F(x) > fmax_F\\
&result_F(x, nearest_F) &&= \textbf{overflow}(-\boldsymbol{\infty}) && \text{if } x \in \mathcal{R} \text{ and } nearest_F(x) < -fmax_F\\
&result_F(x, up_F) &&= \textbf{overflow}(+\boldsymbol{\infty}) && \text{if } x \in \mathcal{R} \text{ and } up_F(x) > fmax_F\\
&result_F(x, up_F) &&= \textbf{overflow}(-fmax_F) && \text{if } x \in \mathcal{R} \text{ and } up_F(x) < -fmax_F\\
&result_F(x, down_F) &&= \textbf{overflow}(fmax_F) && \text{if } x \in \mathcal{R} \text{ and } down_F(x) > fmax_F\\
&result_F(x, down_F) &&= \textbf{overflow}(-\boldsymbol{\infty}) && \text{if } x \in \mathcal{R} \text{ and } down_F(x) < -fmax_F
\end{aligned}$$

For other cases and for any rounding function rnd in $(\mathcal{R} \to F^*)$, the following shall apply:

$$
\begin{array}{lll}
result_F(x, rnd) & = x & \text{if } x = 0 \\
 & = rnd(x) & \text{if } x \in \mathcal{R} \text{ and } fminN_F \leqslant |x| \text{ and } |rnd(x)| \leqslant fmax_F \\
 & = rnd(x) \text{ or } \mathbf{underflow}(c) & \\
 & & \text{if } x \in \mathcal{R} \text{ and } |x| < fminN_F \text{ and } |rnd(x)| = fminN_F \\
 & & \quad \text{and } rnd \text{ has no denormalisation loss at } x \\
 & = rnd(x) \text{ or } \mathbf{underflow}(c) & \\
 & & \text{if } x \in \mathcal{R} \text{ and } denorm_F = \mathbf{true} \text{ and} \\
 & & \quad |rnd(x)| < fminN_F \text{ and } x \neq 0 \\
 & & \quad \text{and } rnd \text{ has no denormalisation loss at } x \\
 & = \mathbf{underflow}(c) & \text{otherwise}
\end{array}
$$

where

$$
\begin{array}{ll}
c = rnd(x) & \text{when } denorm_F = \mathbf{true} \text{ and } (rnd(x) \neq 0 \text{ or } x > 0), \\
c = \mathbf{-0} & \text{when } denorm_F = \mathbf{true} \text{ and } rnd(x) = 0 \text{ and } x < 0, \\
c = 0 & \text{when } denorm_F = \mathbf{false} \text{ and } x > 0, \\
c = \mathbf{-0} & \text{when } denorm_F = \mathbf{false} \text{ and } x < 0
\end{array}
$$

An implementation is allowed to choose between $rnd(x)$ and $\mathbf{underflow}(rnd(x))$ in the region between 0 and $fminN_F$. However, a subnormal value without an underflow notification can be chosen only if $denorm_F = \mathbf{true}$ and no denormalisation loss occurs at x.

NOTES

1 This differs from the specification of $result_F$ as given in part 1 in the following respects: 1) the continuation values on overflow and underflow are given directly here, and 2) all instances of denormalisation loss must be accompanied with an underflow notification.

2 $denorm_F = \mathbf{false}$ implies $iec\_559_F = \mathbf{false}$, and $iec\_559_F = \mathbf{true}$ implies $denorm_F = \mathbf{true}$.

3 If $iec\_559_F = \mathbf{true}$, then subnormal or zero results that have no denormalisation loss do not result in an underflow notification, if the notification is by recording of indicators.

Define the no_result_F, $no\_result2_F$, and $no\_result3_F$ helper functions:

$no\_result_F : F \to \{\mathbf{invalid}\}$

$$
\begin{array}{lll}
no\_result_F(x) & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathbf{qNaN} & \text{if } x \text{ is a quiet NaN} \\
 & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN}
\end{array}
$$

$no\_result2_F : F \times F \to \{\mathbf{invalid}\}$

$$
\begin{array}{lll}
no\_result2_F(x, y) & & \\
 & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x, y \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathbf{qNaN} & \text{if at least one of } x \text{ and } y \text{ is a quiet NaN and} \\
 & & \quad \text{neither a signalling NaN} \\
 & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN or } y \text{ is a signalling NaN}
\end{array}
$$

$no\_result3_F : F \times F \times F \to \{\mathbf{invalid}\}$

$$
\begin{array}{lll}
no\_result3_F(x, y, z) & & \\
 & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x, y, z \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathbf{qNaN} & \text{if at least one of } x,\ y, \text{ and } z \text{ is a quiet NaN and} \\
 & & \quad \text{neither is a signalling NaN}
\end{array}
$$

$$\begin{array}{lll} & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if at least one of } x\text{, } y\text{, or } z \text{ is a signalling NaN} \end{array}$$

These helper functions are used to specify both NaN argument handling and to handle non-NaN-argument cases where **invalid**(**qNaN**) is the appropriate result.

NOTE 4 – The handling of other special values, if available, is left unspecified by this part.

5.2.2 Floating point maximum and minimum

The appropriate return value of the maximum and minimum operations given a quiet **NaN** (**qNaN**) as one of the input values depends on the circumstances for each point of use. Sometimes **qNaN** is the appropriate result, sometimes the non-**NaN** argument is the appropriate result. Therefore, two variants each of the floating point maximum and minimum operations are specified here, and the programmer can decide which one is appropriate to use at each particular place of usage, assuming both variants are included in the binding.

$max_F : F \times F \to F$

$$\begin{array}{lll}
max_F(x, y) & = \max\{x, y\} & \text{if } x, y \in F \\
 & = +\infty & \text{if } x = +\infty \text{ and } y \in F \cup \{-\infty, -\mathbf{0}\} \\
 & = y & \text{if } x = -\mathbf{0} \text{ and } y \in F \text{ and } y \geqslant 0 \\
 & = -\mathbf{0} & \text{if } x = -\mathbf{0} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = -\mathbf{0}) \\
 & = y & \text{if } x = -\infty \text{ and } y \in F \cup \{+\infty, -\mathbf{0}\} \\
 & = +\infty & \text{if } y = +\infty \text{ and } x \in F \cup \{+\infty, -\mathbf{0}\} \\
 & = x & \text{if } y = -\mathbf{0} \text{ and } x \in F \text{ and } x \geqslant 0 \\
 & = -\mathbf{0} & \text{if } y = -\mathbf{0} \text{ and } x \in F \text{ and } x < 0 \\
 & = x & \text{if } y = -\infty \text{ and } x \in F \cup \{-\infty, -\mathbf{0}\} \\
 & = no\_result2_F(x, y) & \text{otherwise}
\end{array}$$

$min_F : F \times F \to F$

$$\begin{array}{lll}
min_F(x, y) & = \min\{x, y\} & \text{if } x, y \in F \\
 & = y & \text{if } x = +\infty \text{ and } y \in F \cup \{-\infty, -\mathbf{0}\} \\
 & = -\mathbf{0} & \text{if } x = -\mathbf{0} \text{ and } y \in F \text{ and } y \geqslant 0 \\
 & = y & \text{if } x = -\mathbf{0} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = -\mathbf{0}) \\
 & = -\infty & \text{if } x = -\infty \text{ and } y \in F \cup \{+\infty, -\mathbf{0}\} \\
 & = x & \text{if } y = +\infty \text{ and } x \in F \cup \{+\infty, -\mathbf{0}\} \\
 & = -\mathbf{0} & \text{if } y = -\mathbf{0} \text{ and } x \in F \text{ and } x \geqslant 0 \\
 & = x & \text{if } y = -\mathbf{0} \text{ and } x \in F \text{ and } x < 0 \\
 & = -\infty & \text{if } y = -\infty \text{ and } x \in F \cup \{-\infty, -\mathbf{0}\} \\
 & = no\_result2_F(x, y) & \text{otherwise}
\end{array}$$

$mmax_F : F \times F \to F$

$$\begin{array}{lll}
mmax_F(x, y) & = max_F(x, y) & \text{if } x, y \in F \cup \{+\infty, -\mathbf{0}, -\infty\} \\
 & = x & \text{if } x \in F \cup \{+\infty, -\mathbf{0}, -\infty\} \text{ and } y \text{ is a quiet NaN} \\
 & = y & \text{if } y \in F \cup \{+\infty, -\mathbf{0}, -\infty\} \text{ and } x \text{ is a quiet NaN} \\
 & = no\_result2_F(x, y) & \text{otherwise}
\end{array}$$

$mmin_F : F \times F \to F$

$$\begin{aligned}
\mathit{mmin}_F(x,y) &= \mathit{min}_F(x,y) && \text{if } x,y \in F \cup \{+\infty, \mathbf{-0}, -\infty\}\\
&= x && \text{if } x \in F \cup \{+\infty, \mathbf{-0}, -\infty\} \text{ and } y \text{ is a quiet NaN}\\
&= y && \text{if } y \in F \cup \{+\infty, \mathbf{-0}, -\infty\} \text{ and } x \text{ is a quiet NaN}\\
&= \mathit{no\_result2}_F(x,y) && \text{otherwise}
\end{aligned}$$

$\mathit{max\_seq}_F : [F] \to F \cup \{\mathbf{infinitary}\}$

$$\begin{aligned}
&\mathit{max\_seq}_F([x_1, ..., x_n])\\
&\quad= \mathbf{infinitary}(-\infty) && \text{if } n = 0\\
&\quad= x_1 && \text{if } n = 1 \text{ and } x_1 \text{ is not a NaN}\\
&\quad= \mathit{max}_F(\mathit{max\_seq}_F([x_1, ..., x_{n-1}]), x_n)\\
& && \text{if } n \geqslant 2\\
&\quad= \mathit{no\_result}_F(x_1) && \text{otherwise}
\end{aligned}$$

$\mathit{min\_seq}_F : [F] \to F \cup \{\mathbf{infinitary}\}$

$$\begin{aligned}
&\mathit{min\_seq}_F([x_1, ..., x_n])\\
&\quad= \mathbf{infinitary}(+\infty) && \text{if } n = 0\\
&\quad= x_1 && \text{if } n = 1 \text{ and } x_1 \text{ is not a NaN}\\
&\quad= \mathit{min}_F(\mathit{min\_seq}_F([x_1, ..., x_{n-1}]), x_n)\\
& && \text{if } n \geqslant 2\\
&\quad= \mathit{no\_result}_F(x_1) && \text{otherwise}
\end{aligned}$$

$\mathit{mmax\_seq}_F : [F] \to F \cup \{\mathbf{infinitary}\}$

$$\begin{aligned}
&\mathit{mmax\_seq}_F([x_1, ..., x_n])\\
&\quad= \mathbf{infinitary}(-\infty) && \text{if } n = 0\\
&\quad= x_1 && \text{if } n = 1 \text{ and } x_1 \text{ is not a NaN}\\
&\quad= \mathit{mmax}_F(\mathit{mmax\_seq}_F([x_1, ..., x_{n-1}]), x_n)\\
& && \text{if } n \geqslant 2\\
&\quad= \mathit{no\_result}_F(x_1) && \text{otherwise}
\end{aligned}$$

$\mathit{mmin\_seq}_F : [F] \to F \cup \{\mathbf{infinitary}\}$

$$\begin{aligned}
&\mathit{mmin\_seq}_F([x_1, ..., x_n])\\
&\quad= \mathbf{infinitary}(+\infty) && \text{if } n = 0\\
&\quad= x_1 && \text{if } n = 1 \text{ and } x_1 \text{ is not a NaN}\\
&\quad= \mathit{mmin}_F(\mathit{mmin\_seq}_F([x_1, ..., x_{n-1}]), x_n)\\
& && \text{if } n \geqslant 2\\
&\quad= \mathit{no\_result}_F(x_1) && \text{otherwise}
\end{aligned}$$

5.2.3 Floating point diminish

$\mathit{dim}_F : F \times F \to F \cup \{\mathbf{underflow}, \mathbf{overflow}\}$

$$\begin{aligned}
\mathit{dim}_F(x,y) &= \mathit{result}_F(\max\{0, x-y)\}, \mathit{rnd}_F)\\
& && \text{if } x,y \in F\\
&= \mathbf{-0} && \text{if } x = \mathbf{-0} \text{ and } y = 0\\
&= \mathit{dim}_F(0,y) && \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{-\infty, \mathbf{-0}, +\infty\} \text{ and } y \neq 0\\
&= \mathit{dim}_F(x,0) && \text{if } y = \mathbf{-0} \text{ and } x \in F \cup \{-\infty, +\infty\}
\end{aligned}$$

$$
\begin{array}{lll}
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \text{ and } y \in F \cup \{\mathbf{-\infty}\} \\
 & = 0 & \text{if } x = \mathbf{-\infty} \text{ and } y \in F \cup \{\mathbf{+\infty}\} \\
 & = 0 & \text{if } y = \mathbf{+\infty} \text{ and } x \in F \\
 & = \mathbf{+\infty} & \text{if } y = \mathbf{-\infty} \text{ and } x \in F \\
 & = \mathit{no\_result2}_F(x, y) & \text{otherwise}
\end{array}
$$

NOTE – dim_F cannot be implemented by $\mathit{max}_F(\mathbf{-0}, \mathit{sub}_F(x, y))$, since this latter expression has other overflow properties.

5.2.4 Floor, round, and ceiling

$\mathit{floor}_F : F \rightarrow F$

$$
\begin{array}{lll}
\mathit{floor}_F(x) & = \lfloor x \rfloor & \text{if } x \in F \\
 & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathit{no\_result}_F(x) & \text{otherwise}
\end{array}
$$

$\mathit{floor\_rest}_F : F \rightarrow F$

$$
\begin{array}{lll}
\mathit{floor\_rest}_F(x) & = \mathit{result}_F(x - \lfloor x \rfloor, \mathit{rnd}_F) & \text{if } x \in F \\
 & = 0 & \text{if } x = \mathbf{-0} \\
 & = \mathit{no\_result}_F(x) & \text{otherwise}
\end{array}
$$

$\mathit{rounding}_F : F \rightarrow F \cup \{\mathbf{-0}\}$

$$
\begin{array}{lll}
\mathit{rounding}_F(x) & = \mathrm{round}(x) & \text{if } x \in F \text{ and } (x \geqslant 0 \text{ or } \mathrm{round}(x) \neq 0) \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x < 0 \text{ and } \mathrm{round}(x) = 0 \\
 & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathit{no\_result}_F(x) & \text{otherwise}
\end{array}
$$

NOTE – round_F is a different operation specified in part 1.

$\mathit{rounding\_rest}_F : F \rightarrow F$

$\mathit{rounding\_rest}_F(x)$

$$
\begin{array}{ll}
= x - \mathrm{round}(x) & \text{if } x \in F \\
= 0 & \text{if } x = \mathbf{-0} \\
= \mathit{no\_result}_F(x) & \text{otherwise}
\end{array}
$$

$\mathit{ceiling}_F : F \rightarrow F \cup \{\mathbf{-0}\}$

$$
\begin{array}{lll}
\mathit{ceiling}_F(x) & = \lceil x \rceil & \text{if } x \in F \text{ and } (x \geqslant 0 \text{ or } \lceil x \rceil \neq 0) \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x < 0 \text{ and } \lceil x \rceil = 0 \\
 & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathit{no\_result}_F(x) & \text{otherwise}
\end{array}
$$

$\mathit{ceiling\_rest}_F : F \rightarrow F$

$$\begin{aligned}
&\mathit{ceiling\_rest}_F(x)\\
&\qquad = \mathit{result}_F(x - \lceil x \rceil, \mathit{rnd}_F) && \text{if } x \in F\\
&\qquad = 0 && \text{if } x = \mathbf{-0}\\
&\qquad = \mathit{no\_result}_F(x) && \text{otherwise}
\end{aligned}$$

5.2.5 Remainder after division with round to integer

$\mathit{residue}_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{invalid}\}$

$$\begin{aligned}
\mathit{residue}_F(x, y) &= \mathit{result}_F(x - (\mathrm{round}(x/y) \cdot y), \mathit{nearest}_F)\\
&&& \text{if } x, y \in F \text{ and } y \neq 0 \text{ and}\\
&&& \quad (x \geqslant 0 \text{ or } x - (\mathrm{round}(x/y) \cdot y) \neq 0)\\
&= \mathbf{-0} && \text{if } x, y \in F \text{ and } y \neq 0 \text{ and}\\
&&& \quad x < 0 \text{ and } x - (\mathrm{round}(x/y) \cdot y) = 0\\
&= \mathbf{-0} && \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } y \neq 0\\
&= x && \text{if } x \in F \text{ and } y \in \{\mathbf{-\infty}, \mathbf{+\infty}\}\\
&= \mathit{no\_result2}_F(x, y) && \text{otherwise}
\end{aligned}$$

5.2.6 Square root and reciprocal square root

$\mathit{sqrt}_F : F \to F \cup \{\mathbf{invalid}\}$

$$\begin{aligned}
\mathit{sqrt}_F(x) &= \mathit{nearest}_F(\sqrt{x}) && \text{if } x \in F \text{ and } x \geqslant 0\\
&= x && \text{if } x \in \{\mathbf{-0}, \mathbf{+\infty}\}\\
&= \mathit{no\_result}_F(x) && \text{otherwise}
\end{aligned}$$

$\mathit{rec\_sqrt}_F : F \to F \cup \{\mathbf{infinitary}, \mathbf{invalid}\}$

$$\begin{aligned}
\mathit{rec\_sqrt}_F(x) &= \mathit{rnd}_F(1/\sqrt{x}) && \text{if } x \in F \text{ and } x > 0\\
&= \mathbf{infinitary}(\mathbf{+\infty}) && \text{if } x \in \{\mathbf{-0}, 0\}\\
&= 0 && \text{if } x = \mathbf{+\infty}\\
&= \mathit{no\_result}_F(x) && \text{otherwise}
\end{aligned}$$

5.2.7 Multiplication to higher precision floating point datatype

For the following operation, F' is a floating point datatype conforming to part 1, where $r_{F'} = r_F$ and $p_{F'} > p_F$.

$\mathit{mul}_{F \to F'} : F \times F \to F' \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}\}$

$\mathit{mul}_{F \to F'}(x, y) = \mathit{mul}_{F'}(\mathit{convert}_{F \to F'}(x), \mathit{convert}_{F \to F'}(y))$

NOTES

1 $\mathit{convert}_{F \to F'}$ is specified in clause 5.4.4.

2 F' has the same radix as, but higher precision than F. If the precision is sufficiently much higher, rounding can be avoided. If also $\mathit{emin}_{F'}$ is sufficiently smaller than emin_F, underflow can be avoided, and if $\mathit{emax}_{F'}$ is sufficiently greater than emax_F, overflow can be avoided.

5.2.8 Support operations for extended floating point precision

These operations are useful when keeping guard digits or implementing extra precision floating point datatypes. The resulting datatypes, e.g. so-called doubled precision, do not necessarily conform to part 1.

$$add\_lo_F : F \times F \rightarrow F \cup \{\mathbf{underflow}\}$$

$$\begin{array}{lll} add\_lo_F(x,y) & = result_F((x+y) - rnd_F(x+y), rnd_F) & \\ & & \text{if } x,y \in F \\ & = \mathbf{-0} & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = \mathbf{-0} & \text{if } x \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } y = \mathbf{-0} \\ & = y & \text{if } x = \mathbf{+\infty} \text{ and } y \in F \cup \{\mathbf{+\infty}\} \\ & = y & \text{if } x = \mathbf{-\infty} \text{ and } y \in F \cup \{\mathbf{-\infty}\} \\ & = x & \text{if } x \in F \text{ and } y \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\ & = no\_result2_F(x,y) & \text{otherwise} \end{array}$$

$$sub\_lo_F : F \times F \rightarrow F \cup \{\mathbf{underflow}\}$$

$$sub\_lo_F(x,y) = add\_lo_F(x, neg_F(y))$$

NOTE 1 – If $rnd\_style_F = nearest$, then, in the absence of notifications, add_lo_F and sub_lo_F return exact results.

$$mul\_lo_F : F \times F \rightarrow F \cup \{\mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{array}{lll} mul\_lo_F(x,y) & = result_F((x \cdot y) - rnd_F(x \cdot y), rnd_F) & \\ & & \text{if } x,y \in F \\ & = mul\_lo_F(0,y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = mul\_lo_F(x,0) & \text{if } x \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } y = \mathbf{-0} \\ & = mul_F(x,y) & \text{if } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \\ & = mul_F(x,y) & \text{if } x \in F \text{ and } y \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\ & = no\_result2_F(x,y) & \text{otherwise} \end{array}$$

NOTE 2 – In the absence of notifications, mul_lo_F returns an exact result.

$$div\_rest_F : F \times F \rightarrow F \cup \{\mathbf{underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll} div\_rest_F(x,y) & = result_F(x - (y \cdot rnd_F(x/y)), rnd_F) & \\ & & \text{if } x,y \in F \\ & = div\_rest_F(0,y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = x & \text{if } x \in F \text{ and } y \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\ & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } y \in F \\ & = no\_result2_F(x,y) & \text{otherwise} \end{array}$$

$$sqrt\_rest_F : F \rightarrow F \cup \{\mathbf{underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll} sqrt\_rest_F(x) & = result_F(x - (sqrt_F(x) \cdot sqrt_F(x)), rnd_F) & \\ & & \text{if } x \in F \text{ and } x \geqslant 0 \\ & = \mathbf{-0} & \text{if } x = \mathbf{-0} \end{array}$$

$$
\begin{aligned}
&= +\infty && \text{if } x = +\infty \\
&= no\_result_F(x) && \text{otherwise}
\end{aligned}
$$

NOTE 3 – $sqrt\_rest_F(x)$ is exact when there is no **underflow**.

5.3 Elementary transcendental floating point operations

The specifications for each of the floating point transcendental operations and the floating point conversion operations (clause 5.4) use an approximation helper function. The approximation helper functions are ideally identical to the true mathematical functions. However, that would imply a maximum error for the corresponding operation of 0.5 ulp (i.e., the minimum value for operations that are not always exact). This part does not require that the maximum error is only 0.5 ulp for the operations specified in clause 5.3, but allows the maximum error to be a bit bigger. To express this, the approximation helper functions need not be identical to the mathematical elementary transcendental functions, but are allowed to be approximate. The approximation helper functions shall be defined for the elements of its given argument signature where the corresponding mathematical function is also defined, unless otherwise noted. The requirements on approximation helper functions apply only where the approximation helper functions are defined.

5.3.1 Maximum error requirements

The approximation helper functions for the individual operations in these subclauses have maximum error parameters that describe the maximum *relative* error, in ulps, of the helper function composed with $nearest_F$, for non-subnormal and non-zero results. The maximum error parameters also describe the maximum *absolute* error, in ulps, for $-fminN_F$, $fminN_F$, subnormal, or zero results and underflow continuation values if $denorm_F =$ **true**. All maximum error parameters shall have a value that is $\geqslant 0.5$. For the maximum value for the maximum error parameters, see the specification of each of the maximum error parameters. See also Annex A, on partial conformity. The relevant maximum error parameters shall be made available to programs.

When the maximum error for an approximation helper function h_F, approximating f, is $max\_error\_op_F$, then for all arguments $x, ... \in F \times ...$ the following equation shall hold:

$$|f(x, ...) - nearest_F(h_F(x, ...))| \leqslant max\_error\_op_F \cdot r_F^{e_F(f(x,...))-p_F}$$

NOTES

1 Partially conforming implementations may have greater values for maximum error parameters than stipulated below. See annex A.

2 For most positive (and not too small) return values t, the true result is thus claimed to be in the interval $[t-(max\_error\_op_F \cdot ulp_F(t)), t+(max\_error\_op_F \cdot ulp_F(t))]$. But if the return value is exactly r_F^n for some not too small $n \in \mathcal{Z}$, then the true result is claimed to be in the interval $[t-(max\_error\_op_F \cdot ulp_F(t)/r_F), t+(max\_error\_op_F \cdot ulp_F(t))]$. Similarly for negative return values.

The results of the approximating helper functions in this clause must be exact for certain arguments as detailed below, and may be exact for all arguments. If the approximating helper function is exact for all arguments, then the corresponding maximum error parameter should have the value 0.5, the minimum value.

5.3.2 Sign requirements

For this part, the approximation helper functions shall be zero exactly at the points where the approximated mathematical function is exactly zero. For this part, at points where the approximation helper functions are not zero, they shall have the same sign as the approximated mathematical function at that point. For the radian trigonometric helper functions, these zero and sign requirements are imposed only for arguments, x, such that $|x| \leqslant big\_angle\_r_F$ (see clause 5.3.8).

NOTE – For the operations, the continuation value after an **underflow** may be zero (including negative zero) as given by $result^*_F$ (see below), even though the approximation helper function is not zero at that point. Such zero results are required to be accompanied by an **underflow** notification. When appropriate, zero may also be returned for IEC 60559 infinities arguments. See the individual specifications.

5.3.3 Monotonicity requirements

For this part, each approximation helper function shall be a monotonic approximation to the mathematical function it is approximating, except:

a) For the radian trigonometric approximation helper functions, the monotonic approximation requirement is imposed only for arguments, x, such that $|x| \leqslant big\_angle\_r_F$ (see clause 5.3.8).

b) The argument angular unit trigonometric and argument angular unit inverse trigonometric approximating helper functions, as well as the angular unit conversion helper functions, are excepted from the monotonic approximation requirement for the angular unit argument(s).

5.3.4 The *result** helper function

The $result^*_F$ helper function is similar to the $result_F$ helper function (see clause 5.2.1), but is simplified compared to $result_F$ concerning **underflow**: $result^*_F$ always underflows for non-zero arguments that have an absolute value less than $fminN_F - (fminD_F/r_F)$, whereas $result_F$ does not necessarily underflow in that case. This difference from $result_F$ is made since the argument to $result^*_F$ might not be exact. To return **underflow** or not, for a tiny result, based upon an inexact argument would be misleading. For the operations specified using $result^*_F$ where the specification implies that there can never be any denormalisation loss for certain tiny results, **underflow** is instead explicitly avoided.

$$result^*_F : \mathcal{R} \times (\mathcal{R} \to F^*) \to F \cup \{\textbf{underflow}, \textbf{overflow}\}$$

$$\begin{aligned} result^*_F(x, rnd) &= \textbf{underflow}(c) && \text{if } x \in \mathcal{R} \text{ and } denorm_F = \textbf{true} \text{ and} \\ & && \quad |rnd(x)| < fminN_F \text{ and } x \neq 0 \\ &= result_F(x, rnd) && \text{otherwise} \end{aligned}$$

where

$$\begin{aligned} c &= rnd(x) && \text{when } rnd(x) \neq 0 \text{ or } x > 0, \\ c &= \mathbf{-0} && \text{when } rnd(x) = 0 \text{ and } x < 0 \end{aligned}$$

5.3.5 Hypotenuse

There shall be a maximum error parameter for the $hypot_F$ operation:

$$max\_error\_hypot_F \in F$$

The $max\_error\_hypot_F$ parameter shall have a value that is $\leqslant 1$.

The $hypot^*_F$ approximation helper function:

$$hypot^*_F : F \times F \rightarrow \mathcal{R}$$

$hypot^*_F(x, y)$ returns a close approximation to $\sqrt{x^2 + y^2}$ in $\mathcal{R}$, with maximum error $max\_error\_hypot_F$.

Further requirements on the $hypot^*_F$ approximation helper function are:

$$\begin{array}{ll}
hypot^*_F(x, y) = hypot^*_F(y, x) & \text{if } x, y \in F \\
hypot^*_F(-x, y) = hypot^*_F(x, y) & \text{if } x, y \in F \\
hypot^*_F(x, y) \geqslant \max\{|x|, |y|\} & \text{if } x, y \in F \\
hypot^*_F(x, y) \leqslant |x| + |y| & \text{if } x, y \in F \\
hypot^*_F(x, y) \geqslant 1 & \text{if } x, y \in F \text{ and } \sqrt{x^2 + y^2} \geqslant 1 \\
hypot^*_F(x, y) \leqslant 1 & \text{if } x, y \in F \text{ and } \sqrt{x^2 + y^2} \leqslant 1
\end{array}$$

The $hypot_F$ operation:

$$hypot_F : F \times F \rightarrow F \cup \{\mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{array}{lll}
hypot_F(x, y) & = result^*_F(hypot^*_F(x, y), nearest_F) & \\
 & & \text{if } x, y \in F \\
 & = hypot_F(0, y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = hypot_F(x, 0) & \text{if } y = \mathbf{-0} \text{ and } x \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
 & = \mathbf{+\infty} & \text{if } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
 & = \mathbf{+\infty} & \text{if } y \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } x \in F \\
 & = no\_result2_F(x, y) & \text{otherwise}
\end{array}$$

5.3.6 Operations for exponentiations and logarithms

There shall be two maximum error parameters for approximate exponentiations and logarithms:

$$max\_error\_exp_F \in F$$
$$max\_error\_power_F \in F$$

The $max\_error\_exp_F$ parameter shall have a value that is $\leqslant 1.5 \cdot rnd\_error_F$. The $max\_error\_power_F$ parameter shall have a value that is $\leqslant 2 \cdot rnd\_error_F$.

5.3.6.1 Integer power of argument base

The $power^*_{F,I}$ approximation helper function:

$$power^*_{F,I} : F \times I \rightarrow \mathcal{R}$$

$power^*_{F,I}(x, y)$ returns a close approximation to x^y in $\mathcal{R}$, with maximum error $max\_error\_power_F$.

Further requirements on the $power^*_{F,I}$ approximation helper function are:

$power^*_{F,I}(x, y) = x^y$	if $x \in \mathcal{Z} \cap F$ and $y \in I$ and ($\lvert x\rvert = 1$ or $y > 0$)
$power^*_{F,I}(x, 1) = x$	if $x \in F$
$power^*_{F,I}(x, 0) = 1$	if $x \in F$ and $x \neq 0$
$power^*_{F,I}(x, y) < fminD_F/2$	if $x \in F$ and $x > 0$ and $y \in I$ and $x^y < fminD_F/3$
$power^*_{F,I}(x, y) = power^*_{F,I}(-x, y)$	if $x \in F$ and $x < 0$ and $y \in I$ and $2\lvert y$
$power^*_{F,I}(x, y) = -power^*_{F,I}(-x, y)$	if $x \in F$ and $x < 0$ and $y \in I$ and not $2\lvert y$

The relationship to other $power^*_{FI'}$ helper functions for any $power_{FI'}$ operations in the same library shall be:

$power^*_{FI}(x, y) = power^*_{FI'}(x, y)$ if $x \in F$ and $y \in I \cap I'$

The $power_{FI}$ operation:

$power_{FI} : F \times I \to F \cup \{\textbf{underflow}, \textbf{overflow}, \textbf{infinitary}\}$

$power_{FI}(x, y)$	$= result^*_F(power^*_{FI}(x, y), nearest_F)$	
		if $x \in F$ and $x \neq 0$ and $y \in I$
	$= \mathbf{+\infty}$	if $x = \mathbf{-\infty}$ and $y \in I$ and $y > 0$ and $2\lvert y$
	$= \mathbf{-\infty}$	if $x = \mathbf{-\infty}$ and $y \in I$ and $y > 0$ and not $2\lvert y$
	$= 0$	if $x = \mathbf{-0}$ and $y \in I$ and $y > 0$ and $2\lvert y$
	$= \mathbf{-0}$	if $x = \mathbf{-0}$ and $y \in I$ and $y > 0$ and not $2\lvert y$
	$= 0$	if $x = 0$ and $y \in I$ and $y > 0$
	$= \mathbf{+\infty}$	if $x = \mathbf{+\infty}$ and $y \in I$ and $y > 0$
	$= 1$	if $x \in \{\mathbf{-\infty}, \mathbf{-0}, 0, \mathbf{+\infty}\}$ and $y = 0$
	$= 0$	if $x = \mathbf{-\infty}$ and $y \in I$ and $y < 0$ and $2\lvert y$
	$= \mathbf{-0}$	if $x = \mathbf{-\infty}$ and $y \in I$ and $y < 0$ and not $2\lvert y$
	$= \textbf{infinitary}(\mathbf{+\infty})$	if $x = \mathbf{-0}$ and $y \in I$ and $y < 0$ and $2\lvert y$
	$= \textbf{infinitary}(\mathbf{-\infty})$	if $x = \mathbf{-0}$ and $y \in I$ and $y < 0$ and not $2\lvert y$
	$= \textbf{infinitary}(\mathbf{+\infty})$	if $x = 0$ and $y \in I$ and $y < 0$
	$= 0$	if $x = \mathbf{+\infty}$ and $y \in I$ and $y < 0$
	$= no\_result_F(x)$	otherwise

NOTES

1 $power_{FI}(x, y)$ will overflow approximately when $x^y > fmax_F$, i.e., if $x > 1$, approximately when $y > \log_x(fmax_F)$, and if $0 < x < 1$, approximately when $y < \log_x(fmax_F)$ (which is then negative). It will not overflow when $x = 0$ or when $x = 1$.

2 $power_I$ (in clause 5.1.4) does not allow most negative *exponents* (unless $\lvert x\rvert = 1$) since the exact result then is not in $\mathcal{Z}$ unless $\lvert x\rvert = 1$. $power_F$ (in clause 5.3.6.6) does not allow any negative *bases* since the (exact) result is not in $\mathcal{R}$ unless the exponent is integer. $power_{FI}$ takes care of this latter case, where all exponents are ensured to be integers that have not arisen from implicit floating point rounding.

5.3.6.2 Natural exponentiation

The exp^*_F approximation helper function:

$exp^*_F : F \to \mathcal{R}$

$exp^*_F(x)$ returns a close approximation to e^x in $\mathcal{R}$, with maximum error $max\_error\_exp_F$.

Further requirements on the exp^*_F approximation helper function are:

$exp^*_F(1) = e$

$exp^*_F(x) = 1$ if $x \in F$ and $exp^*_F(x) \neq e^x$ and $\ln(1 - (epsilon_F/(2 \cdot r_F))) < x$ and $x < \ln(1 + (epsilon_F/2))$

$exp^*_F(x) < fminD_F/2$ if $x \in F$ and $x < \ln(fminD_F) - 3$

The exp_F operation:

$exp_F : F \to F \cup \{\textbf{underflow}, \textbf{overflow}\}$

$exp_F(x)$	$= result^*_F(exp^*_F(x), nearest_F)$	if $x \in F$
	$= 1$	if $x = \mathbf{-0}$
	$= \mathbf{+\infty}$	if $x = \mathbf{+\infty}$
	$= 0$	if $x = \mathbf{-\infty}$
	$= no\_result_F(x)$	otherwise

NOTES

1 $exp_F(1) = nearest_F(e)$.

2 $exp_F(x)$ will overflow approximately when $x > \ln(fmax_F)$.

5.3.6.3 Natural exponentiation, minus one

The $expm1^*_F$ approximation helper function:

$expm1^*_F : F \to \mathcal{R}$

$expm1^*_F(x)$ returns a close approximation to $e^x - 1$ in $\mathcal{R}$, with maximum error $max\_error\_exp_F$.

Further requirements on the $expm1^*_F$ approximation helper function are:

$expm1^*_F(1) = e - 1$

$expm1^*_F(x) = x$ if $x \in F$ and $expm1^*_F(x) \neq e^x - 1$ and $-epsilon_F/r_F \leqslant x < 0.5 \cdot epsilon_F/r_F$

$expm1^*_F(x) = -1$ if $x \in F$ and $expm1^*_F(x) \neq e^x - 1$ and $x < \ln(epsilon_F/(3 \cdot r_F))$

The relationship to the exp^*_F approximation helper function for the exp_F operation in the same library shall be:

$expm1^*_F(x) \leqslant exp^*_F(x)$ if $x \in F$

The $expm1_F$ operation:

$expm1_F : F \to F \cup \{\textbf{overflow}\}$

$expm1_F(x)$	$= result^*_F(expm1^*_F(x), nearest_F)$	if $x \in F$ and $\lvert x\rvert \geqslant fminN_F$
	$= x$	if $x \in F$ and $\lvert x\rvert < fminN_F$
	$= \mathbf{-0}$	if $x = \mathbf{-0}$
	$= \mathbf{+\infty}$	if $x = \mathbf{+\infty}$
	$= -1$	if $x = \mathbf{-\infty}$
	$= no\_result_F(x)$	otherwise

NOTES

1 **underflow** is explicitly avoided. Part 1 requires that $\mathit{fminN}_F \leqslant \mathit{epsilon}_F$. This part requires that $\mathit{fminN}_F < 0.5 \cdot \mathit{epsilon}_F / r_F$, so that underflow can be avoided here.

2 $\mathit{expm1}_F(1) = \mathit{nearest}_F(e-1)$.

3 $\mathit{expm1}_F(x)$ will overflow approximately when $x > \ln(\mathit{fmax}_F)$.

5.3.6.4 Exponentiation of 2

The $\mathit{exp2}^*_F$ approximation helper function:

$$\mathit{exp2}^*_F : F \to \mathcal{R}$$

$\mathit{exp2}^*_F(x)$ returns a close approximation to 2^x in $\mathcal{R}$, with maximum error $\mathit{max\_error\_exp}_F$.

Further requirements on the $\mathit{exp2}^*_F$ approximation helper function are:

$$\begin{array}{ll}
\mathit{exp2}^*_F(x) = 1 & \text{if } x \in F \text{ and } \mathit{exp2}^*_F(x) \neq 2^x \text{ and} \\
 & \quad \log_2(1 - (\mathit{epsilon}_F/(2 \cdot r_F))) < x \text{ and} \\
 & \quad x < \log_2(1 + (\mathit{epsilon}_F/2)) \\
\mathit{exp2}^*_F(x) = 2^x & \text{if } x \in F \cap \mathcal{Z} \text{ and } 2^x \in F \\
\mathit{exp2}^*_F(x) < \mathit{fminD}_F/2 & \text{if } x \in F \text{ and } x < \log_2(\mathit{fminD}_F) - 3
\end{array}$$

The $\mathit{exp2}_F$ operation:

$$\mathit{exp2}_F : F \to F \cup \{\textbf{underflow}, \textbf{overflow}\}$$

$$\begin{array}{lll}
\mathit{exp2}_F(x) & = \mathit{result}^*_F(\mathit{exp2}^*_F(x), \mathit{nearest}_F) & \\
 & & \text{if } x \in F \\
 & = 1 & \text{if } x = \mathbf{-0} \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \\
 & = 0 & \text{if } x = \mathbf{-\infty} \\
 & = \mathit{no\_result}_F(x) & \text{otherwise}
\end{array}$$

NOTE – $\mathit{exp2}_F(x)$ will overflow approximately when $x > \log_2(\mathit{fmax}_F)$.

5.3.6.5 Exponentiation of 10

The $\mathit{exp10}^*_F$ approximation helper function:

$$\mathit{exp10}^*_F : F \to \mathcal{R}$$

$\mathit{exp10}^*_F(x)$ returns a close approximation to 10^x in $\mathcal{R}$, with maximum error $\mathit{max\_error\_exp}_F$.

Further requirements on the $\mathit{exp10}^*_F$ approximation helper function are:

$$\begin{array}{ll}
\mathit{exp10}^*_F(x) = 1 & \text{if } x \in F \text{ and } \mathit{exp10}^*_F(x) \neq 10^x \text{ and} \\
 & \quad \log_{10}(1 - (\mathit{epsilon}_F/(2 \cdot r_F))) < x \text{ and} \\
 & \quad x < \log_{10}(1 + (\mathit{epsilon}_F/2)) \\
\mathit{exp10}^*_F(x) = 10^x & \text{if } x \in F \cap \mathcal{Z} \text{ and } 10^x \in F \\
\mathit{exp10}^*_F(x) < \mathit{fminD}_F/2 & \text{if } x \in F \text{ and } x < \log_{10}(\mathit{fminD}_F) - 3
\end{array}$$

The $\mathit{exp10}_F$ operation:

$$\mathit{exp10}_F : F \to F \cup \{\textbf{underflow}, \textbf{overflow}\}$$

$$
\begin{aligned}
exp10_F(x) &= result^*_F(exp10^*_F(x), nearest_F) && \text{if } x \in F \\
&= 1 && \text{if } x = \mathbf{-0} \\
&= \mathbf{+\infty} && \text{if } x = \mathbf{+\infty} \\
&= 0 && \text{if } x = \mathbf{-\infty} \\
&= no\_result_F(x) && \text{otherwise}
\end{aligned}
$$

NOTE – $exp10_F(x)$ will overflow approximately when $x > \log_{10}(fmax_F)$.

5.3.6.6 Exponentiation of argument base

The $power^*_F$ approximation helper function:

$power^*_F : F \times F \to \mathcal{R}$

$power^*_F(x, y)$ returns a close approximation to x^y in $\mathcal{R}$, with maximum error $max\_error\_power_F$. The $power^*_F$ helper function need be defined only for first arguments that are greater than 0.

Further requirements on the $power^*_F$ approximation helper function are:

$$
\begin{aligned}
&power^*_F(1, y) = 1 && \text{if } y \in F \\
&power^*_F(x, 0) = 1 && \text{if } x \in F \text{ and } x > 0 \\
&power^*_F(x, 1) = x && \text{if } x \in F \text{ and } x > 0 \\
&power^*_F(x, y) < fminD_F/2 && \text{if } x \in F \text{ and } x > 0 \text{ and } y \in F \text{ and } x^y < fminD_F/3
\end{aligned}
$$

The relationship to the $power^*_{FI}$ approximation helper functions for any $power_{FI}$ operations in the same library shall be:

$$power^*_F(x, y) = power^*_{F,I}(x, y) \qquad \text{if } x \in F \text{ and } x > 0 \text{ and } y \in I \cap F$$

The $power_F$ operation:

$power_F : F \times F \to F \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\}$

$$
\begin{aligned}
power_F(x, y) &= result^*_F(power^*_F(x, y), nearest_F) && \text{if } x \in F \text{ and } x > 0 \text{ and } y \in F \\
&= power_F(0, y) && \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
&= power_F(x, 0) && \text{if } y = \mathbf{-0} \text{ and } x \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
&= \mathbf{+\infty} && \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y > 0) \text{ or } y = \mathbf{+\infty}) \\
&= \mathbf{+\infty} && \text{if } x \in F \text{ and } x > 1 \text{ and } y = \mathbf{+\infty} \\
&= 0 && \text{if } x \in F \text{ and } 0 \leqslant x < 1 \text{ and } y = \mathbf{+\infty} \\
&= 0 && \text{if } x = 0 \text{ and } y \in F \text{ and } y > 0 \\
&= \mathbf{infinitary}(\mathbf{+\infty}) && \text{if } x = 0 \text{ and } y \in F \text{ and } y < 0 \\
&= \mathbf{+\infty} && \text{if } x \in F \text{ and } 0 \leqslant x < 1 \text{ and } y = \mathbf{-\infty} \\
&= 0 && \text{if } x \in F \text{ and } x > 1 \text{ and } y = \mathbf{-\infty} \\
&= 0 && \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-\infty}) \\
&= no\_result2_F(x, y) && \text{otherwise}
\end{aligned}
$$

NOTE – $power_F(x, y)$ will overflow approximately when $x^y > fmax_F$, i.e., if $x > 1$, approximately when $y > \log_x(fmax_F)$, and if $0 < x < 1$, approximately when $y < \log_x(fmax_F)$ (which is a negative number). It will not overflow when $x = 0$ or when $x = 1$.

5.3.6.7 Exponentiation of one plus the argument base, minus one

The $\mathit{power1pm1}^*_F$ approximation helper function:

$$\mathit{power1pm1}^*_F : F \times F \to \mathcal{R}$$

$\mathit{power1pm1}^*_F(x, y)$ returns a close approximation to $(1 + x)^y - 1$ in $\mathcal{R}$, with maximum error $\mathit{max\_error\_power}_F$. The $\mathit{power1pm1}^*_F$ helper function need be defined only for first arguments that are greater than or equal to -1.

Further requirements on the $\mathit{power1pm1}^*_F$ approximation helper function are:

$$\begin{array}{ll}
\mathit{power1pm1}^*_F(x,y) = (1+x)^y - 1 & \text{if } x, y \in F \cap \mathcal{Z} \text{ and } x \geqslant -1 \text{ and } y > 0 \\
\mathit{power1pm1}^*_F(x,1) = x & \text{if } x, 1+x \in F \text{ and } x > -1 \\
\mathit{power1pm1}^*_F(-1,y) = -1 & \text{if } y \in F \text{ and } y > 0 \\
\mathit{power1pm1}^*_F(x,y) = -1 & \text{if } x \in F \text{ and } x > -1 \text{ and } y \in F \text{ and} \\
 & \quad \mathit{power1pm1}^*_F(x,y) \neq (1+x)^y - 1 \text{ and} \\
 & \quad (1+x)^y < \mathit{epsilon}_F/(3 \cdot r_F)
\end{array}$$

The relationship to the power^*_F approximation helper function for the power_F operation in the same library shall be:

$$\mathit{power1pm1}^*_F(x,y) \leqslant \mathit{power}^*_F(1+x, y) \qquad \text{if } x, 1+x \in F \text{ and } x > -1 \text{ and } y \in F$$

NOTE 1 – $\mathit{power1pm1}^*_F(x,y) \approx y \cdot \ln(1+x)$ if $x \in F$ and $x > -1$ and $y \in F$ and $|y \cdot \ln(1+x)| < \mathit{epsilon}_F/r_F$.

The $\mathit{power1pm1}_F$ operation:

$$\mathit{power1pm1}_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
\mathit{power1pm1}_F(x,y) & & \\
 & = \mathit{result}^*_F(\mathit{power1pm1}^*_F(x,y), \mathit{nearest}_F) & \\
 & & \text{if } x \in F \text{ and } x > -1 \text{ and } x \neq 0 \text{ and } y \in F \text{ and } y \neq 0 \\
 & = \mathit{mul}_F(x,y) & \text{if } x \in \{\mathbf{-0}, 0\} \text{ and } y \in F \text{ and } y \neq 0 \\
 & = \mathit{mul}_F(x,y) & \text{if } y \in \{\mathbf{-0}, 0\} \text{ and } x \in F \text{ and } x > -1 \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y > 0) \text{ or } y = \mathbf{+\infty}) \\
 & = \mathbf{+\infty} & \text{if } x \in F \text{ and } x > 0 \text{ and } y = \mathbf{+\infty} \\
 & = -1 & \text{if } x \in F \text{ and } -1 \leqslant x < 0 \text{ and } y = \mathbf{+\infty} \\
 & = -1 & \text{if } x = -1 \text{ and } y \in F \text{ and } y > 0 \\
 & = \mathbf{infinitary}(\mathbf{+\infty}) & \text{if } x = -1 \text{ and } y \in F \text{ and } y < 0 \\
 & = \mathbf{+\infty} & \text{if } x \in F \text{ and } -1 \leqslant x < 0 \text{ and } y = \mathbf{-\infty} \\
 & = -1 & \text{if } x \in F \text{ and } x > 0 \text{ and } y = \mathbf{-\infty} \\
 & = -1 & \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-\infty}) \\
 & & \\
 & = \mathit{no\_result2}_F(x,y) & \text{otherwise}
\end{array}$$

NOTE 2 – $\mathit{power1pm1}_F(x,y)$ will overflow approximately when $(1 + x)^y > \mathit{fmax}_F$, i.e., if $x > 0$, approximately when $y > \log_{1+x}(\mathit{fmax}_F)$, and if $-1 < x < 0$, approximately when $y < \log_{1+x}(\mathit{fmax}_F)$. It will not overflow when $x \in \{-1, 0\}$.

5.3.6.8 Natural logarithm

The ln^*_F approximation helper function:

$$\mathit{ln}^*_F : F \cup \{e\} \to \mathcal{R}$$

$ln^*_F(x)$ returns a close approximation to $\ln(x)$ in $\mathcal{R}$, with maximum error $max\_error\_exp_F$.

A further requirement on the ln^*_F approximation helper function is:

$ln^*_F(e) = 1$

The ln_F operation:

$ln_F : F \to F \cup \{\textbf{infinitary}, \textbf{invalid}\}$

$$\begin{aligned}
ln_F(x) &= result^*_F(ln^*_F(x), nearest_F) && \\
&&& \text{if } x \in F \text{ and } x > 0 \\
&= \textbf{infinitary}(-\infty) && \text{if } x \in \{\mathbf{-0}, 0\} \\
&= \mathbf{+\infty} && \text{if } x = \mathbf{+\infty} \\
&= no\_result_F(x) && \text{otherwise}
\end{aligned}$$

5.3.6.9 Natural logarithm of one plus the argument

The $ln1p^*_F$ approximation helper function:

$ln1p^*_F : F \cup \{e - 1\} \to \mathcal{R}$

$ln1p^*_F(x)$ returns a close approximation to $\ln(1 + x)$ in $\mathcal{R}$, with maximum error $max\_error\_exp_F$.

Further requirements on the $ln1p^*_F$ approximation helper function are:

$$\begin{aligned}
&ln1p^*_F(e - 1) = 1 && \\
&ln1p^*_F(x) = x && \text{if } x \in F \text{ and } ln1p^*_F(x) \neq \ln(1 + x) \text{ and} \\
&&& \quad -0.5 \cdot epsilon_F/r_F < x \leqslant epsilon_F/r_F
\end{aligned}$$

The relationship to the ln^*_F approximation helper function for the ln_F operation in the same library shall be:

$ln1p^*_F(x) \geqslant ln^*_F(x)$ if $x \in F$ and $x > 0$

The $ln1p_F$ operation:

$ln1p_F : F \to F \cup \{\textbf{infinitary}, \textbf{invalid}\}$

$$\begin{aligned}
ln1p_F(x) &= result^*_F(ln1p^*_F(x), nearest_F) && \\
&&& \text{if } x \in F \text{ and } x > -1 \text{ and } |x| \geqslant fminN_F \\
&= x && \text{if } x \in F \text{ and } |x| < fminN_F \\
&= \mathbf{-0} && \text{if } x = \mathbf{-0} \\
&= \textbf{infinitary}(-\infty) && \text{if } x = -1 \\
&= \mathbf{+\infty} && \text{if } x = \mathbf{+\infty} \\
&= no\_result_F(x) && \text{otherwise}
\end{aligned}$$

NOTE – **underflow** is explicitly avoided. Part 1 requires that $fminN_F \leqslant epsilon_F$. This part requires that $fminN_F < 0.5 \cdot epsilon_F/r_F$, so that underflow can be avoided here.

5.3.6.10 2-logarithm

The $log2^*_F$ approximation helper function:

$log2^*_F : F \to \mathcal{R}$

$log2^*_F(x)$ returns a close approximation to $\log_2(x)$ in $\mathcal{R}$, with maximum error $max\_error\_exp_F$.

A further requirement on the $log2^*_F$ approximation helper function is:

$$\mathit{log2}^*_F(x) = \log_2(x) \qquad \text{if } x \in F \text{ and } \log_2(x) \in \mathcal{Z}$$

The $\mathit{log2}_F$ operation:

$$\mathit{log2}_F : F \to F \cup \{\textbf{infinitary}, \textbf{invalid}\}$$

$$\begin{array}{lll}
\mathit{log2}_F(x) & = \mathit{result}^*_F(\mathit{log2}^*_F(x), \mathit{nearest}_F) & \\
 & & \text{if } x \in F \text{ and } x > 0 \\
 & = \textbf{infinitary}(-\infty) & \text{if } x \in \{\mathbf{-0}, 0\} \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \\
 & = \mathit{no\_result}_F(x) & \text{otherwise}
\end{array}$$

5.3.6.11 10-logarithm

The $\mathit{log10}^*_F$ approximation helper function:

$$\mathit{log10}^*_F : F \to \mathcal{R}$$

$\mathit{log10}^*_F(x)$ returns a close approximation to $\log_{10}(x)$ in $\mathcal{R}$, with maximum error $\mathit{max\_error\_exp}_F$.

A further requirement on the $\mathit{log10}^*_F$ approximation helper function is:

$$\mathit{log10}^*_F(x) = \log_{10}(x) \qquad \text{if } x \in F \text{ and } \log_{10}(x) \in \mathcal{Z}$$

The $\mathit{log10}_F$ operation:

$$\mathit{log10}_F : F \to F \cup \{\textbf{infinitary}, \textbf{invalid}\}$$

$$\begin{array}{lll}
\mathit{log10}_F(x) & = \mathit{result}^*_F(\mathit{log10}^*_F(x), \mathit{nearest}_F) & \\
 & & \text{if } x \in F \text{ and } x > 0 \\
 & = \textbf{infinitary}(-\infty) & \text{if } x \in \{\mathbf{-0}, 0\} \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \\
 & = \mathit{no\_result}_F(x) & \text{otherwise}
\end{array}$$

5.3.6.12 Argument base logarithm

The $\mathit{logbase}^*_F$ approximation helper function:

$$\mathit{logbase}^*_F : F \times F \to \mathcal{R}$$

$\mathit{logbase}^*_F(x, y)$ returns a close approximation to $\log_x(y)$ in $\mathcal{R}$, with maximum error $\mathit{max\_error\_power}_F$.

A further requirement on the $\mathit{logbase}^*_F$ approximation helper function is:

$$\mathit{logbase}^*_F(x, x) = 1 \qquad \text{if } x \in F \text{ and } x > 0 \text{ and } x \neq 1$$

The $\mathit{logbase}_F$ operation:

$$\mathit{logbase}_F : F \times F \to F \cup \{\mathbf{-0}, \textbf{infinitary}, \textbf{invalid}\}$$

$$\begin{array}{lll}
\mathit{logbase}_F(x, y) & = \mathit{result}^*_F(\mathit{logbase}^*_F(x, y), \mathit{nearest}_F) & \\
 & & \text{if } x \in F \text{ and } x > 0 \text{ and } x \neq 1 \text{ and } y \in F \text{ and } y > 0 \\
 & = \mathit{logbase}_F(0, y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathit{logbase}_F(x, 0) & \text{if } y = \mathbf{-0} \text{ and } x \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
 & & \\
 & = \textbf{infinitary}(\mathbf{+\infty}) & \text{if } x = 1 \text{ and } y \in F \text{ and } y > 1 \\
 & = \textbf{infinitary}(\mathbf{-\infty}) & \text{if } x = 1 \text{ and } y \in F \text{ and } 0 \leqslant y < 1 \\
 & = 0 & \text{if } x = \mathbf{+\infty} \text{ and } y \in F \text{ and } y \geqslant 1 \\
 & = \mathbf{+\infty} & \text{if } x \in F \text{ and } 1 \leqslant x \text{ and } y = \mathbf{+\infty}
\end{array}$$

$= -\infty$	if $x \in F$ and $0 < x < 1$ and $y = +\infty$
$= -\mathbf{0}$	if $x = 0$ and $y \in F$ and $y \geqslant 1$
$= 0$	if $x = 0$ and $y \in F$ and $0 < y < 1$
$= \mathbf{infinitary}(+\infty)$	if $x \in F$ and $0 < x < 1$ and $y = 0$
$= \mathbf{infinitary}(-\infty)$	if $x \in F$ and $1 < x$ and $y = 0$
$= -\mathbf{0}$	if $x = +\infty$ and $y \in F$ and $0 < y < 1$
$= no\_result2_F(x, y)$	otherwise

5.3.6.13 Argument base logarithm of one plus each argument

The $logbase1p1p^*_F$ approximation helper function:

$logbase1p1p^*_F : F \times F \to \mathcal{R}$

$logbase1p1p^*_F(x, y)$ returns a close approximation to $\log_{(1+x)}(1 + y)$ in $\mathcal{R}$, with maximum error $max\_error\_power_F$.

A further requirements on $logbase1p1p^*_F$ approximation helper function is:

$logbase1p1p^*_F(x, x) = 1$ if $x \in F$ and $x > -1$ and $x \neq 0$

The $logbase1p1p_F$ operation:

$logbase1p1p_F : F \times F \to F \cup \{-\mathbf{0}, \mathbf{underflow}, \mathbf{infinitary}, \mathbf{invalid}\}$

$logbase1p1p_F(x, y)$

$= result^*_F(logbase1p1p^*_F(x, y), nearest_F)$	
	if $x \in F$ and $x > -1$ and $x \neq 0$ and $y \in F$ and $y > -1$ and $y \neq 0$
$= div_F(y, x)$	if $x \in \{-\mathbf{0}, 0\}$ and $((y \in F$ and $y > -1$ and $y \neq 0)$ or $y = +\infty)$
$= div_F(y, x)$	if $y \in \{-\mathbf{0}, 0\}$ and $((x \in F$ and $x > -1)$ or $x = +\infty)$
$= 0$	if $x = +\infty$ and $y \in F$ and $y \geqslant 0$
$= +\infty$	if $x \in F$ and $0 < x$ and $y = +\infty$
$= -\infty$	if $x \in F$ and $-1 < x < 0$ and $y = +\infty$
$= -\mathbf{0}$	if $x = -1$ and $y \in F$ and $y \geqslant 0$
$= 0$	if $x = -1$ and $y \in F$ and $-1 < y < 0$
$= \mathbf{infinitary}(+\infty)$	if $x \in F$ and $-1 < x < 0$ and $y = -1$
$= \mathbf{infinitary}(-\infty)$	if $x \in F$ and $0 < x$ and $y = -1$
$= -\mathbf{0}$	if $x = +\infty$ and $y \in F$ and $-1 < y < 0$
$= no\_result2_F(x, y)$	otherwise

5.3.7 Introduction to operations for trigonometric elementary functions

Two different operations for each of sin, cos, tan, cot, sec, csc, arcsin, arccos, arctan, arccot, arccotc, arcsec, and arccsc are specified. One version for radians and one version where the angular unit is given as a parameter.

For use in the specifications below, define the following mathematical functions:

$rad : \mathcal{R} \rightarrow \mathcal{R}$
$axis\_rad : \mathcal{R} \rightarrow \{(1,0),(0,1),(-1,0),(0,-1)\} \times \mathcal{R}$
$arc : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$

The *rad*, angular value normalisation, function is defined by

$$rad(x) = x - \mathrm{round}(x/(2\cdot\pi))\cdot 2\cdot\pi$$

The *axis_rad* function is defined by

$$\begin{aligned} axis\_rad(x) &= ((1,0), \arcsin(\sin(x))) && \text{if } \cos(x) \geqslant 1/\sqrt{2} \\ &= ((0,1), \arcsin(\cos(x))) && \text{if } \sin(x) > 1/\sqrt{2} \\ &= ((-1,0), \arcsin(\sin(x))) && \text{if } \cos(x) \leqslant -1/\sqrt{2} \\ &= ((0,-1), \arcsin(\cos(x))) && \text{if } \sin(x) < -1/\sqrt{2} \end{aligned}$$

The *arc*, angle, function is defined by

$$\begin{aligned} arc(x,y) &= -\arccos(x/\sqrt{x^2+y^2}) && \text{if } y < 0 \\ &= \arccos(x/\sqrt{x^2+y^2}) && \text{if } y \geqslant 0 \end{aligned}$$

5.3.8 Operations for radian trigonometric elementary functions

There shall be one radian big-angle parameter:

$big\_angle\_r_F \in F$

It should have the following default value:

$big\_angle\_r_F = r_F^{\lceil p_F/2 \rceil}$

A binding or implementation can include a method to change the value of the radian big-angle parameter. This method should only allow the value of this parameter to be set to a value greater than $2\cdot\pi$ and such that $ulp_F(big\_angle\_r_F) < \pi/1000$.

NOTE – Part 1 requires only that $p_F \geqslant 2$, but see also A.5.2.0.2 in part 1. This part requires that $p_F \geqslant 2\cdot\max\{1, \lceil \log_{r_F}(2\cdot\pi)\rceil\}$, in order to allow at least the first two cycles (plus and minus) to be in the interval $[-big\_angle\_r_F, big\_angle\_r_F]$. In order to allow $ulp_F(big\_angle\_r_F) < \pi/1000$, $p_F \geqslant 2 + \lceil \log_{r_F}(1000)\rceil$ should hold.

For use in the approximation helper function's signatures, define

$$F^{2\cdot\pi} = (F \cup \{n\cdot\pi/4, n\cdot\pi/6 \mid n \in \mathcal{Z}\}) \cap [-big\_angle\_r_F, big\_angle\_r_F]$$

There shall be three maximum error parameters for radian trigonometric operations:

$max\_error\_rad_F \in F$
$max\_error\_sin_F \in F$
$max\_error\_tan_F \in F$

The $max\_error\_rad_F$ parameter shall have a value that is 0.5 (ulp). The $max\_error\_sin_F$ parameter shall have a value that is $\leqslant 1.5\cdot rnd\_error_F$. The $max\_error\_tan_F$ parameter shall have a value that is $\leqslant 2\cdot rnd\_error_F$. If the binding standard requires that the $max\_error\_rad_F$ parameter has the value 0.5, that parameter need not be made available for programs.

5.3.8.1 Radian angle normalisation

The rad^*_F approximation helper function:

$rad^*_F : F^{2\cdot\pi} \to \mathcal{R}$

$rad^*_F(x)$ returns a close approximation to $rad(x)$ in $\mathcal{R}$, if $|x| \leqslant big\_angle\_r_F$, with maximum error $max\_error\_rad_F$.

The $axis\_rad^*_F$ approximation helper function:

$axis\_rad^*_F : F^{2\cdot\pi} \to \{(1,0),(0,1),(-1,0),(0,-1)\} \times \mathcal{R}$

$axis\_rad^*_F(x)$ returns a close approximation to $axis\_rad(x)$, if $|x| \leqslant big\_angle\_r_F$, with maximum error $max\_error\_rad_F$ for the second part of the result. The approximation consists of that the second part of the result (the offset from the indicated axis) is approximate. The first part (the nearest axis indication) shall be exact if $|x| \leqslant big\_angle\_r_F$.

NOTE 1 – With the maximum error 0.5 ulp, these helper functions are not really needed. However, Annex A allows for partial conformity, such that the maximum error for these two helper functions may be greater than 0.5 ulp.

Further requirements on the rad^*_F and $axis\_rad^*_F$ approximation helper functions are:

$rad^*_F(x) = x$ if $x \in F^{2\cdot\pi}$ and $|x| < \pi$

$snd(axis\_rad^*_F(x)) = rad^*_F(x)$ if $x \in F^{2\cdot\pi}$ and $fst(axis\_rad^*_F(x)) = (1,0)$

The rad_F operation:

$rad_F : F \to F \cup \{\mathbf{underflow}, \mathbf{absolute\_precision\_underflow}\}$

$rad_F(x) = result^*_F(rad^*_F(x), nearest_F)$
  if $x \in F$ and $|x| > fminN_F$ and $|x| \leqslant big\_angle\_r_F$
$= x$ if ($x \in F$ and $|x| \leqslant fminN_F$) or $x = \mathbf{-0}$

$= \mathbf{absolute\_precision\_underflow}(\mathbf{qNaN})$
  if $x \in F$ and $|x| > big\_angle\_r_F$
$= no\_result_F(x)$ otherwise

The $axis\_rad_F$ operation:

$axis\_rad_F : F \to (\{(1,0),(0,1),(-1,0),(0,-1)\} \times F) \cup \{\mathbf{absolute\_precision\_underflow}\}$

$axis\_rad_F(x) = (fst(axis\_rad^*_F(x)), result^*_F(snd(axis\_rad^*_F(x)), nearest_F))$
  if $x \in F$ and $|x| > fminN_F$ and $|x| \leqslant big\_angle\_r_F$
$= ((1,0), x)$ if ($x \in F$ and $|x| \leqslant fminN_F$) or $x = \mathbf{-0}$

$= \mathbf{absolute\_precision\_underflow}((\mathbf{qNaN}, \mathbf{qNaN}), \mathbf{qNaN})$
  if $x \in F$ and $|x| > big\_angle\_r_F$
$= ((\mathbf{qNaN}, \mathbf{qNaN}), \mathbf{qNaN})$
  if x is a quiet NaN
$= \mathbf{invalid}((\mathbf{qNaN}, \mathbf{qNaN}), \mathbf{qNaN})$
  otherwise

NOTE 2 – rad_F is simpler, easier to use, but less accurate than $axis\_rad_F$. The latter may still not be sufficient for implementing the radian trigonometric operations to less than the maximum error stated by the parameters.

5.3.8.2 Radian sine

The sin^*_F approximation helper function:

$sin^*_F : F^{2\cdot\pi} \rightarrow \mathcal{R}$

$sin^*_F(x)$ returns a close approximation to $\sin(x)$ in $\mathcal{R}$ if $|x| \leqslant big\_angle\_r_F$, with maximum error $max\_error\_sin_F$.

Further requirements on the sin^*_F approximation helper function are:

$sin^*_F(n \cdot 2 \cdot \pi + \pi/6) = 1/2$ if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi + \pi/6| \leqslant big\_angle\_r_F$
$sin^*_F(n \cdot 2 \cdot \pi + \pi/2) = 1$ if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi + \pi/2| \leqslant big\_angle\_r_F$
$sin^*_F(n \cdot 2 \cdot \pi + 5 \cdot \pi/6) = 1/2$ if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi + 5 \cdot \pi/6| \leqslant big\_angle\_r_F$
$sin^*_F(x) = x$ if $x \in F^{2\cdot\pi}$ and $sin^*_F(x) \neq \sin(x)$ and $|x| \leqslant \sqrt{3 \cdot epsilon_F/r_F}$
$sin^*_F(-x) = -sin^*_F(x)$ if $x \in F^{2\cdot\pi}$

The sin_F operation:

$sin_F : F \rightarrow F \cup \{\textbf{underflow}, \textbf{absolute\_precision\_underflow}\}$

$sin_F(x) = result^*_F(sin^*_F(x), nearest_F)$
if $x \in F$ and $fminN_F < |x|$ and $|x| \leqslant big\_angle\_r_F$
$= rad_F(x)$ otherwise

NOTE – **underflow** is here explicitly avoided for subnormal arguments, but the operation may **underflow** for other arguments.

5.3.8.3 Radian cosine

The cos^*_F approximation helper function:

$cos^*_F : F^{2\cdot\pi} \rightarrow \mathcal{R}$

$cos^*_F(x)$ returns a close approximation to $\cos(x)$ in $\mathcal{R}$ if $|x| \leqslant big\_angle\_r_F$, with maximum error $max\_error\_sin_F$.

Further requirements on the cos^*_F approximation helper function are:

$cos^*_F(n \cdot 2 \cdot \pi) = 1$ if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi| \leqslant big\_angle\_r_F$
$cos^*_F(n \cdot 2 \cdot \pi + \pi/3) = 1/2$ if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi + \pi/3| \leqslant big\_angle\_r_F$
$cos^*_F(n \cdot 2 \cdot \pi + 2 \cdot \pi/3) = -1/2$ if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi + 2 \cdot \pi/3| \leqslant big\_angle\_r_F$
$cos^*_F(n \cdot 2 \cdot \pi + \pi) = -1$ if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi + \pi| \leqslant big\_angle\_r_F$
$cos^*_F(x) = 1$ if $x \in F^{2\cdot\pi}$ and $cos^*_F(x) \neq \cos(x)$ and $|x| < \sqrt{epsilon_F/r_F}$
$cos^*_F(-x) = cos^*_F(x)$ if $x \in F^{2\cdot\pi}$

The cos_F operation:

$cos_F : F \rightarrow F \cup \{\textbf{underflow}, \textbf{absolute\_precision\_underflow}\}$

$cos_F(x) = result^*_F(cos^*_F(x), nearest_F)$
if $x \in F$ and $|x| \leqslant big\_angle\_r_F$
$= 1$ if $x = \mathbf{-0}$
$= rad_F(x)$ otherwise

5.3.8.4 Radian tangent

The tan^*_F approximation helper function:

$$tan^*_F : F^{2\cdot\pi} \rightarrow \mathcal{R}$$

$tan^*_F(x)$ returns a close approximation to $\tan(x)$ in $\mathcal{R}$ if $|x| \leqslant big\_angle\_r_F$, with maximum error $max\_error\_tan_F$.

Further requirements on the tan^*_F approximation helper function are:

$$\begin{array}{ll}
tan^*_F(n\cdot 2\cdot\pi + \pi/4) = 1 & \text{if } n \in \mathcal{Z} \text{ and } |n\cdot 2\cdot\pi + \pi/4| \leqslant big\_angle\_r_F \\
tan^*_F(n\cdot 2\cdot\pi + 3\cdot\pi/4) = -1 & \text{if } n \in \mathcal{Z} \text{ and } |n\cdot 2\cdot\pi + 3\cdot\pi/4| \leqslant big\_angle\_r_F \\
tan^*_F(x) = x & \text{if } x \in F^{2\cdot\pi} \text{ and } tan^*_F(x) \neq \tan(x) \text{ and} \\
 & \quad |x| < \sqrt{epsilon_F/r_F} \\
tan^*_F(-x) = -tan^*_F(x) & \text{if } x \in F^{2\cdot\pi}
\end{array}$$

NOTE 1 – tan has a smallest period of π, but the above expresses a period of $2\cdot\pi$, which is more in line with the other operations. The desired points of extra accuracy are still covered.

The tan_F operation:

$$tan_F : F \rightarrow F \cup \{\textbf{underflow}, \textbf{overflow}, \textbf{absolute\_precision\_underflow}\}$$

$$\begin{array}{lll}
tan_F(x) & = result^*_F(tan^*_F(x), nearest_F) & \\
 & & \text{if } x \in F \text{ and } fminN_F < |x| \text{ and } |x| \leqslant big\_angle\_r_F \\
 & = rad_F(x) & \text{otherwise}
\end{array}$$

NOTE 2 – **underflow** is explicitly avoided for subnormal arguments, but the operation may **underflow** for other arguments.

5.3.8.5 Radian cotangent

The cot^*_F approximation helper function:

$$cot^*_F : F^{2\cdot\pi} \rightarrow \mathcal{R}$$

$cot^*_F(x)$ returns a close approximation to $\cot(x)$ in $\mathcal{R}$ if $|x| \leqslant big\_angle\_r_F$, with maximum error $max\_error\_tan_F$.

Further requirements on the cot^*_F approximation helper function are:

$$\begin{array}{ll}
cot^*_F(n\cdot 2\cdot\pi + \pi/4) = 1 & \text{if } n \in \mathcal{Z} \text{ and } |n\cdot 2\cdot\pi + \pi/4| \leqslant big\_angle\_r_F \\
cot^*_F(n\cdot 2\cdot\pi + 3\cdot\pi/4) = -1 & \text{if } n \in \mathcal{Z} \text{ and } |n\cdot 2\cdot\pi + 3\cdot\pi/4| \leqslant big\_angle\_r_F \\
cot^*_F(-x) = -cot^*_F(x) & \text{if } x \in F^{2\cdot\pi}
\end{array}$$

NOTE – cot has a smallest period of π, but the above expresses a period of $2\cdot\pi$, which is more in line with the other operations. The desired points of extra accuracy are still covered.

The cot_F operation:

$$cot_F : F \rightarrow F \cup \{\textbf{underflow}, \textbf{overflow}, \textbf{infinitary}, \textbf{absolute\_precision\_underflow}\}$$

$$\begin{array}{lll}
cot_F(x) & = result^*_F(cot^*_F(x), nearest_F) & \\
 & & \text{if } x \in F \text{ and } x \neq 0 \text{ and } |x| \leqslant big\_angle\_r_F \\
 & = \textbf{infinitary}(+\infty) & \text{if } x = 0 \\
 & = \textbf{infinitary}(-\infty) & \text{if } x = \mathbf{-0} \\
 & = rad_F(x) & \text{otherwise}
\end{array}$$

5.3.8.6 Radian secant

The sec^*_F approximation helper function:

$$sec^*_F : F^{2\cdot\pi} \to \mathcal{R}$$

$sec^*_F(x)$ returns a close approximation to $\sec(x)$ in $\mathcal{R}$ if $|x| \leqslant big\_angle\_r_F$, with maximum error $max\_error\_tan_F$.

Further requirements on the sec^*_F approximation helper function are:

$sec^*_F(n \cdot 2 \cdot \pi) = 1$	if $n \in \mathcal{Z}$ and $\lvert n \cdot 2 \cdot \pi\rvert \leqslant big\_angle\_r_F$
$sec^*_F(n \cdot 2 \cdot \pi + \pi/3) = 2$	if $n \in \mathcal{Z}$ and $\lvert n \cdot 2 \cdot \pi + \pi/3\rvert \leqslant big\_angle\_r_F$
$sec^*_F(n \cdot 2 \cdot \pi + 2 \cdot \pi/3) = -2$	if $n \in \mathcal{Z}$ and $\lvert n \cdot 2 \cdot \pi + 2 \cdot \pi/3\rvert \leqslant big\_angle\_r_F$
$sec^*_F(n \cdot 2 \cdot \pi + \pi) = -1$	if $n \in \mathcal{Z}$ and $\lvert n \cdot 2 \cdot \pi + \pi\rvert \leqslant big\_angle\_r_F$
$sec^*_F(x) = 1$	if $x \in F^{2\cdot\pi}$ and $sec^*_F(x) \neq \sec(x)$ and $\lvert x\rvert < \sqrt{epsilon_F}$
$sec^*_F(-x) = sec^*_F(x)$	if $x \in F^{2\cdot\pi}$

The sec_F operation:

$$sec_F : F \to F \cup \{\mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$$

$$\begin{aligned} sec_F(x) &= result^*_F(sec^*_F(x), nearest_F) && \text{if } x \in F \text{ and } |x| \leqslant big\_angle\_r_F \\ &= 1 && \text{if } x = \mathbf{-0} \\ &= rad_F(x) && \text{otherwise} \end{aligned}$$

5.3.8.7 Radian cosecant

The csc^*_F approximation helper function:

$$csc^*_F : F^{2\cdot\pi} \to \mathcal{R}$$

$csc^*_F(x)$ returns a close approximation to $\csc(x)$ in $\mathcal{R}$ if $|x| \leqslant big\_angle\_r_F$, with maximum error $max\_error\_tan_F$.

Further requirements on the csc^*_F approximation helper function are:

$csc^*_F(n \cdot 2 \cdot \pi + \pi/6) = 2$	if $n \in \mathcal{Z}$ and $\lvert n \cdot 2 \cdot \pi + \pi/6\rvert \leqslant big\_angle\_r_F$
$csc^*_F(n \cdot 2 \cdot \pi + \pi/2) = 1$	if $n \in \mathcal{Z}$ and $\lvert n \cdot 2 \cdot \pi + \pi/2\rvert \leqslant big\_angle\_r_F$
$csc^*_F(n \cdot 2 \cdot \pi + 5 \cdot \pi/6) = 2$	if $n \in \mathcal{Z}$ and $\lvert n \cdot 2 \cdot \pi + 5 \cdot \pi/6\rvert \leqslant big\_angle\_r_F$
$csc^*_F(-x) = -csc^*_F(x)$	if $x \in F^{2\cdot\pi}$

The csc_F operation:

$$csc_F : F \to F \cup \{\mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}\}$$

$$\begin{aligned} csc_F(x) &= result^*_F(csc^*_F(x), nearest_F) && \text{if } x \in F \text{ and } x \neq 0 \text{ and } |x| \leqslant big\_angle\_r_F \\ &= \mathbf{infinitary}(+\infty) && \text{if } x = 0 \\ &= \mathbf{infinitary}(-\infty) && \text{if } x = \mathbf{-0} \\ &= rad_F(x) && \text{otherwise} \end{aligned}$$

5.3.8.8 Radian cosine with sine

$$\begin{aligned}
&cossin_F : F \to (F \times F) \cup \{\textbf{underflow}, \textbf{absolute\_precision\_underflow}\}\\
&cossin_F(x) \quad = (cos_F(x), sin_F(x))
\end{aligned}$$

NOTES

1 If there is an **absolute_precision_underflow** notification, then both result parts suffer from the **absolute_precision_underflow** and the continuation values for both parts are **qNaN**. Similarly for NaN and infinitary arguments.

2 If there is an **underflow** notification, only one of the result parts suffer from the underflow, and the other part has an absolute value greater than $fminN_F$.

5.3.8.9 Radian arc sine

The $arcsin^*_F$ approximation helper function:

$$arcsin^*_F : F \to \mathcal{R}$$

$arcsin^*_F(x)$ returns a close approximation to $\arcsin(x)$ in $\mathcal{R}$, with maximum error $max\_error\_sin_F$.

Further requirements on the $arcsin^*_F$ approximation helper function are:

$$\begin{aligned}
&arcsin^*_F(1/2) = \pi/6\\
&arcsin^*_F(1) = \pi/2\\
&arcsin^*_F(x) = x && \text{if } x \in F \text{ and } arcsin^*_F(x) \neq \arcsin(x) \text{ and}\\
& && \quad |x| < \sqrt{2 \cdot epsilon_F / r_F}\\
&arcsin^*_F(-x) = -arcsin^*_F(x) && \text{if } x \in F
\end{aligned}$$

The $arcsin^\#_F$ range limitation helper function (for $x \in F$):

$$arcsin^\#_F(x) = \max\{up_F(-\pi/2), \min\{arcsin^*_F(x), down_F(\pi/2)\}\}$$

The $arcsin_F$ operation:

$$arcsin_F : F \to F \cup \{\textbf{invalid}\}$$

$$\begin{aligned}
arcsin_F(x) \quad &= result^*_F(arcsin^\#_F(x), nearest_F)\\
& \qquad\qquad \text{if } x \in F \text{ and } fminN_F < |x| \leqslant 1\\
&= x \qquad\qquad \text{if } (x \in F \text{ and } |x| \leqslant fminN_F) \text{ or } x = \mathbf{-0}\\
&= no\_result_F(x) \qquad \text{otherwise}
\end{aligned}$$

NOTE – **underflow** is explicitly avoided.

5.3.8.10 Radian arc cosine

The $arccos^*_F$ approximation helper function:

$$arccos^*_F : F \to \mathcal{R}$$

$arccos^*_F(x)$ returns a close approximation to $\arccos(x)$ in $\mathcal{R}$, with maximum error $max\_error\_sin_F$.

Further requirements on the $arccos^*_F$ approximation helper function are:

$arccos^*_F(1/2) = \pi/3$
$arccos^*_F(0) = \pi/2$
$arccos^*_F(-1/2) = 2 \cdot \pi/3$
$arccos^*_F(-1) = \pi$

The $arccos^\#_F$ range limitation helper function (for $x \in F$):

$arccos^\#_F(x) = \min\{arccos^*_F(x), down_F(\pi)\}$

The $arccos_F$ operation:

$arccos_F : F \rightarrow F \cup \{\mathbf{invalid}\}$

$$\begin{array}{lll} arccos_F(x) & = result^*_F(arccos^\#_F(x), nearest_F) & \\ & & \text{if } x \in F \text{ and } |x| \leqslant 1 \\ & = arccos_F(0) & \text{if } x = \mathbf{-0} \\ & = no\_result_F(x) & \text{otherwise} \end{array}$$

5.3.8.11 Radian arc tangent

The $arctan^*_F$ approximation helper function:

$arctan^*_F : F \rightarrow \mathcal{R}$

$arctan^*_F(x)$ returns a close approximation to $\arctan(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tan_F$.

Further requirements on the $arctan^*_F$ approximation helper function are:

$$\begin{array}{ll} arctan^*_F(1) = \pi/4 & \\ arctan^*_F(x) = x & \text{if } x \in F \text{ and } arctan^*_F(x) \neq \arctan(x) \text{ and} \\ & \quad |x| \leqslant \sqrt{1.5 \cdot epsilon_F/r_F} \\ arctan^*_F(x) = \pi/2 & \text{if } x \in F \text{ and } arctan^*_F(x) \neq \arctan(x) \text{ and} \\ & \quad x > 3 \cdot r_F/epsilon_F \\ arctan^*_F(-x) = -arctan^*_F(x) & \text{if } x \in F \end{array}$$

The $arctan^\#_F$ range limitation helper function (for $x \in F$):

$arctan^\#_F(x) = \max\{up_F(-\pi/2), \min\{arctan^*_F(x), down_F(\pi/2)\}\}$

The $arctan_F$ operation:

$arctan_F : F \rightarrow F$

$$\begin{array}{lll} arctan_F(x) & = result^*_F(arctan^\#_F(x), nearest_F) & \\ & & \text{if } x \in F \text{ and } fminN_F < |x| \\ & = x & \text{if } (x \in F \text{ and } |x| \leqslant fminN_F) \text{ or } x = \mathbf{-0} \\ & = up_F(-\pi/2) & \text{if } x = \mathbf{-\infty} \\ & = down_F(\pi/2) & \text{if } x = \mathbf{+\infty} \\ & = no\_result_F(x) & \text{otherwise} \end{array}$$

NOTES

1 $arctan_F(x) \approx arc_F(1, x)$. ($arc_F$ is specified in subclause 5.3.8.15 below.)

2 **underflow** is explicitly avoided.

5.3.8.12 Radian arc cotangent

This clause specifies two inverse cotangent operations. One approximating the sign symmetric (but discontinuous at 0) arccot, the other approximating the continuous (but not sign symmetric) arccotc.

The $arccot^*_F$ approximation helper function:

$arccot^*_F : F \to \mathcal{R}$

$arccot^*_F(x)$ returns a close approximation to $\text{arccot}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tan_F$.

The $arccotc^*_F$ approximation helper function:

$arccotc^*_F : F \to \mathcal{R}$

$arccotc^*_F(x)$ returns a close approximation to $\text{arccotc}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tan_F$.

Further requirements on the $arccot^*_F$ and $arccotc^*_F$ approximation helper functions are:

$arccot^*_F(1) = \pi/4$
$arccot^*_F(0) = \pi/2$
$arccot^*_F(-x) = -arccot^*_F(x)$ if $x \in F$ and $x \neq 0$

$arccotc^*_F(x) = arccot^*_F(x)$ if $x \in F$ and $x \geqslant 0$
$arccotc^*_F(-1) = 3 \cdot \pi/4$
$arccotc^*_F(x) = \pi$ if $x \in F$ and $arccotc^*_F(x) \neq \text{arccotc}(x)$ and $x < -3 \cdot r_F/epsilon_F$

The $arccot^{\#}_F$ and $arccotc^{\#}_F$ range limitation helper functions (for $x \in F$):

$arccot^{\#}_F(x) = \max\{up_F(-\pi/2), \min\{arccot^*_F(x), down_F(\pi/2)\}\}$
$arccotc^{\#}_F(x) = \min\{arccotc^*_F(x), down_F(\pi)\}$

The $arccot_F$ operation:

$arccot_F : F \to F \cup \{\textbf{underflow}\}$

$$
\begin{aligned}
arccot_F(x) &= result^*_F(arccot^{\#}_F(x), nearest_F) && \\
&&& \text{if } x \in F \\
&= up_F(-\pi/2) && \text{if } x = \mathbf{-0} \\
&= \mathbf{-0} && \text{if } x = \mathbf{-\infty} \\
&= 0 && \text{if } x = \mathbf{+\infty} \\
&= no\_result_F(x) && \text{otherwise}
\end{aligned}
$$

NOTES

1 $arccot_F(neg_F(x)) = neg_F(arccot_F(x))$.

2 Due to the range limitation, $arccot_F(0)$ need not equal $arccotc_F(0)$.

The $arccotc_F$ operation:

$arccotc_F : F \to F \cup \{\textbf{underflow}\}$

$$
\begin{aligned}
arccotc_F(x) &= result^*_F(arccotc^{\#}_F(x)) && \\
&&& \text{if } x \in F \\
&= nearest_F(\pi/2) && \text{if } x = \mathbf{-0} \\
&= down_F(\pi) && \text{if } x = \mathbf{-\infty} \\
&= 0 && \text{if } x = \mathbf{+\infty} \\
&= no\_result_F(x) && \text{otherwise}
\end{aligned}
$$

NOTE 3 – $arccotc_F(x) \approx arc_F(x, 1)$. ($arc_F$ is specified in subclause 5.3.8.15 below.)

5.3.8.13 Radian arc secant

The $arcsec^*_F$ approximation helper function:

$$arcsec^*_F : F \to \mathcal{R}$$

$arcsec^*_F(x)$ returns a close approximation to $\operatorname{arcsec}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tan_F$.

Further requirements on the $arcsec^*_F$ approximation helper function are:

$$\begin{array}{ll}
arcsec^*_F(2) = \pi/3 & \\
arcsec^*_F(-2) = 2 \cdot \pi/3 & \\
arcsec^*_F(-1) = \pi & \\
arcsec^*_F(x) \leqslant \pi/2 & \text{if } x \in F \text{ and } x > 0 \\
arcsec^*_F(x) \geqslant \pi/2 & \text{if } x \in F \text{ and } x < 0 \\
arcsec^*_F(x) = \pi/2 & \text{if } x \in F \text{ and } arcsec^*_F(x) \neq \operatorname{arcsec}(x) \text{ and} \\
 & \quad |x| > 3 \cdot r_F/epsilon_F
\end{array}$$

The $arcsec^{\#}_F$ range limitation helper function (for $x \in F$):

$$\begin{array}{lll}
arcsec^{\#}_F(x) & = \min\{arcsec^*_F(x), down_F(\pi/2)\} & \\
 & & \text{if } x \geqslant 1 \\
 & = \max\{up_F(\pi/2), \min\{arcsec^*_F(x), down_F(\pi)\}\} & \\
 & & \text{if } x \leqslant -1
\end{array}$$

The $arcsec_F$ operation:

$$arcsec_F : F \to F \cup \{\textbf{invalid}\}$$

$$\begin{array}{lll}
arcsec_F(x) & = result^*_F(arcsec^{\#}_F(x), nearest_F) & \\
 & & \text{if } x \in F \text{ and } 1 \leqslant |x| \\
 & = up_F(\pi/2) & \text{if } x = -\infty \\
 & = down_F(\pi/2) & \text{if } x = +\infty \\
 & = no\_result_F(x) & \text{otherwise}
\end{array}$$

5.3.8.14 Radian arc cosecant

The $arccsc^*_F$ approximation helper function:

$$arccsc^*_F : F \to \mathcal{R}$$

$arccsc^*_F(x)$ returns a close approximation to $\operatorname{arccsc}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tan_F$.

Further requirements on the $arccsc^*_F$ approximation helper function are:

$$\begin{array}{ll}
arccsc^*_F(2) = \pi/6 & \\
arccsc^*_F(1) = \pi/2 & \\
arccsc^*_F(-x) = -arccsc^*_F(x) & \text{if } x \in F
\end{array}$$

The $arccsc^{\#}_F$ range limitation helper function (for $x \in F$):

$$arccsc^{\#}_F(x) = \max\{up_F(-\pi/2), \min\{arccsc^*_F(x), down_F(\pi/2)\}\}$$

The $arccsc_F$ operation:

$$arccsc_F : F \to F \cup \{\textbf{underflow}, \textbf{invalid}\}$$

$arccsc_F(x)$	$= result^*_F(arccsc^\#_F(x), nearest_F)$	
		if $x \in F$ and $1 \leqslant \lvert x \rvert$
	$= \mathbf{-0}$	if $x = \mathbf{-\infty}$
	$= 0$	if $x = \mathbf{+\infty}$
	$= no\_result_F(x)$	otherwise

5.3.8.15 Radian angle from Cartesian co-ordinates

The arc^*_F approximation helper function:

$arc^*_F : F \times F \to \mathcal{R}$

$arc^*_F(x, y)$ returns a close approximation to $arc(x, y)$ in $\mathcal{R}$, with maximum error $max\_error\_tan_F$.

NOTE 1 – The arc operations, with the arguments swapped, are often called `arctan2`.

Further requirements on the arc^*_F approximation helper function are:

$arc^*_F(x, 0) = 0$	if $x \in F$ and $x > 0$
$arc^*_F(x, x) = \pi/4$	if $x \in F$ and $x > 0$
$arc^*_F(0, y) = \pi/2$	if $y \in F$ and $y > 0$
$arc^*_F(x, -x) = 3 \cdot \pi/4$	if $x \in F$ and $x < 0$
$arc^*_F(x, 0) = \pi$	if $x \in F$ and $x < 0$
$arc^*_F(x, -y) = -arc^*_F(x, y)$	if $x, y \in F$ and ($y \neq 0$ or $x > 0$)

The $arc^\#_F$ range limitation helper function (for $x, y \in F$):

$arc^\#_F(x, y) = \max\{up_F(-\pi), \min\{arc^*_F(x, y), down_F(\pi)\}\}$

The arc_F operation:

$arc_F : F \times F \to F \cup \{\textbf{underflow}\}$

$arc_F(x, y)$	$= result^*_F(arc^\#_F(x, y), nearest_F)$	
		if $x, y \in F$ and ($x \neq 0$ or $y \neq 0$)
	$= 0$	if $x = 0$ and $y = 0$
	$= down_F(\pi)$	if $x = \mathbf{-0}$ and $y = 0$
	$= arc_F(0, y)$	if $x = \mathbf{-0}$ and $y \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\}$ and $y \neq 0$
	$= neg_F(arc_F(x, 0))$	if $y = \mathbf{-0}$ and $x \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\}$
	$= 0$	if $x = \mathbf{+\infty}$ and $y \in F$ and $y \geqslant 0$
	$= \mathbf{-0}$	if $x = \mathbf{+\infty}$ and $y \in F$ and $y < 0$
	$= nearest_F(\pi/4)$	if $x = \mathbf{+\infty}$ and $y = \mathbf{+\infty}$
	$= nearest_F(\pi/2)$	if $x \in F$ and $y = \mathbf{+\infty}$
	$= nearest_F(3 \cdot \pi/4)$	if $x = \mathbf{-\infty}$ and $y = \mathbf{+\infty}$
	$= down_F(\pi)$	if $x = \mathbf{-\infty}$ and $y \in F$ and $y \geqslant 0$
	$= up_F(-\pi)$	if $x = \mathbf{-\infty}$ and $y \in F$ and $y < 0$
	$= nearest_F(-3 \cdot \pi/4)$	if $x = \mathbf{-\infty}$ and $y = \mathbf{-\infty}$
	$= nearest_F(-\pi/2)$	if $x \in F$ and $y = \mathbf{-\infty}$
	$= nearest_F(-\pi/4)$	if $x = \mathbf{+\infty}$ and $y = \mathbf{-\infty}$
	$= no\_result2_F(x, y)$	otherwise

NOTE 2 – Note that the arc operations do *not* return an **invalid** notification at the origin (both arguments in $\{\mathbf{-0}, 0\}$). See B.5.3.8. Bindings may choose to alter this behaviour.

5.3.9 Operations for trigonometrics with given angular unit

There shall be one big-angle parameter for argument angular-unit trigonometric operations:

$$big\_angle\_u_F \in F$$

It should have the following default value:

$$big\_angle\_u_F = \lceil r_F^{\lceil p_F/2 \rceil}/6 \rceil$$

A binding or implementation can include a method to change the value of this parameter. This method should only allow the value of this parameter to be set to a value greater than or equal to 1 and such that $ulp_F(big\_angle\_u_F) \leqslant 1/2000$.

NOTE 1 – In order to allow $ulp_F(big\_angle\_u_F) \leqslant 1/2000$, $p_F \geqslant 2 + \log_{r_F}(1000)$ should hold.

There shall be a derived parameter signifying the minimum allowed angular unit:

$$min\_angular\_unit_F = r_F \cdot fminN_F/epsilon_F$$

NOTE 2 – That is, $min\_angular\_unit_F = r_F^{(emin_F - 1 + p_F)}$

For use in the approximation helper function's signatures, define

$$F^u = (F \cup \{n \cdot u/8, n \cdot u/12 \mid n \in \mathcal{Z}\}) \cap [-big\_angle\_u_F \cdot |u|, big\_angle\_u_F \cdot |u|]$$

Note that u is a parameter here, a parameter which is the value of the first argument to the approximation helper function. To signify this, the notation $(u : F)$ is used below.

To make the specifications below a bit easier to express, let

$$G_F = \{x \in F \mid min\_angular\_unit_F \leqslant |x|\}.$$

Let $T = \{1, 2, 360, 400, 6400\}$. T consists of angle values for exactly one revolution for some common non-radian angular units: cycles, half-cycles, arc degrees, grades, and mils.

There shall be two parameterised maximum error parameters for argument angular-unit trigonometric operations:

$$max\_error\_sinu_F : F \to F \cup \{\mathbf{invalid}\}$$
$$max\_error\_tanu_F : F \to F \cup \{\mathbf{invalid}\}$$

For $u \in G_F$, the $max\_error\_sinu_F(u)$ parameter shall have a value that is $\leqslant 2 \cdot max\_error\_sin_F$. The $max\_error\_sinu_F(u)$ parameter shall have the value of $max\_error\_sin_F$ if $|u| \in T$. For $u \in G_F$, the $max\_error\_tanu_F(u)$ parameter shall have a value that is $\leqslant 2 \cdot max\_error\_tan_F$. The $max\_error\_tanu_F(u)$ parameter shall have the value of $max\_error\_tan_F$ if $|u| \in T$. The $max\_error\_sinu_F(u)$ and $max\_error\_tanu_F(u)$ parameters return **invalid(qNaN)** if $u \notin G_F$.

5.3.9.1 Argument angular-unit angle normalisation

The argument angular-unit normalisation computes exactly $rad(2 \cdot \pi \cdot x/u) \cdot u/(2 \cdot \pi)$, where x is the angular value, and u is the angular unit.

The $cycle_F$ operation:

$$cycle_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{aligned}
cycle_F(u, x) &= residue_F(x, u) && \text{if } u \in G_F \text{ and } (x = \mathbf{-0} \text{ or} \\
&&& \quad (x \in F \text{ and } |x/u| \leqslant big\_angle\_u_F)) \\
&= \mathbf{absolute\_precision\_underflow}(\mathbf{qNaN}) \\
&&& \text{if } u \in G_F \text{ and } x \in F \text{ and } |x/u| > big\_angle\_u_F \\
&= no\_result2_F(u, x) && \text{otherwise}
\end{aligned}$$

The $axis\_cycle_F$ operation:

$$axis\_cycle_F : F \times F \to (\{(1,0),(0,1),(-1,0),(0,-1)\} \times (F \cup \{\mathbf{-0}\})) \cup \{\mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{aligned}
axis\_cycle_F(u,x) & \\
&= (axis(u,x), result_F(x - (\mathrm{round}(x \cdot 4/u) \cdot u/4), nearest_F)) && \text{if } u \in G_F \text{ and } x \in F \text{ and } |x/u| \leqslant big\_angle\_u_F \text{ and} \\
& && \quad (x/u \geqslant 0 \text{ or } x - (\mathrm{round}(x \cdot 4/u) \cdot u/4) \neq 0) \\
&= (axis(u,x), \mathbf{-0}) && \text{if } u \in G_F \text{ and } x \in F \text{ and } |x/u| \leqslant big\_angle\_u_F \text{ and} \\
& && \quad x/u < 0 \text{ and } x - (\mathrm{round}(x \cdot 4/u) \cdot u/4) = 0 \\
&= ((1,0), \mathbf{-0}) && \text{if } u \in G_F \text{ and } x = \mathbf{-0} \\
&= \mathbf{absolute\_precision\_underflow}((\mathbf{qNaN}, \mathbf{qNaN}), \mathbf{qNaN}) && \\
& && \text{if } u \in G_F \text{ and } x \in F \text{ and } |x/u| > big\_angle\_u_F \\
&= ((\mathbf{qNaN}, \mathbf{qNaN}), \mathbf{qNaN}) && \\
& && \text{if at least one of } x \text{ and } u \text{ is a quiet NaN and} \\
& && \quad \text{neither is a signalling NaN} \\
&= \mathbf{invalid}((\mathbf{qNaN}, \mathbf{qNaN}), \mathbf{qNaN}) && \\
& && \text{otherwise}
\end{aligned}$$

where

$$\begin{aligned}
axis(u,x) &= (1,0) && \text{if } \mathrm{round}(x \cdot 4/u) = 4 \cdot n \\
&= (0,1) && \text{if } \mathrm{round}(x \cdot 4/u) = 4 \cdot n + 1 \\
&= (-1,0) && \text{if } \mathrm{round}(x \cdot 4/u) = 4 \cdot n + 2 \\
&= (0,-1) && \text{if } \mathrm{round}(x \cdot 4/u) = 4 \cdot n + 3
\end{aligned}$$

for some $n \in \mathcal{Z}$.

NOTES

1 $axis\_cycle_F(u,x)$ is exact when $div_F(u,4) = u/4$.

2 $cycle_F$ is an exact operation.

3 $cycle_F(u,x)$ is $\mathbf{-0}$ or has a result in the interval $[-|u/2|, |u/2|]$ if there is no notification.

4 A zero resulting angle is negative if the original angle value is negative.

5 The $cycle_F$ operation is used also in the specifications of the unit argument trigonometric operations. This does *not* imply that the implementation has to use the $cycle_F$ operation, when implementing the operations. It only implies that the *results* (including notifications) must be *as if* the $cycle_F$ operation was used.

5.3.9.2 Argument angular-unit sine

The $sinu^*_F$ approximation helper function:

$$sinu^*_F : (u : F) \times F^u \to \mathcal{R}$$

$sinu^*_F(u,x)$ returns a close approximation to $\sin(x \cdot 2 \cdot \pi/u)$ in $\mathcal{R}$ if $u \neq 0$, with maximum error $max\_error\_sinu_F(u)$.

Further requirements on the $sinu^*_F$ approximation helper function are:

$$\begin{aligned}
sinu^*_F(u, n \cdot u + x) &= sinu^*_F(u,x) && \text{if } n \in \mathcal{Z} \text{ and } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
sinu^*_F(u, u/12) &= 1/2 && \text{if } u \in F \text{ and } u \neq 0 \\
sinu^*_F(u, u/4) &= 1 && \text{if } u \in F \text{ and } u \neq 0
\end{aligned}$$

$$\begin{array}{ll}
sinu^*_F(u, 5 \cdot u/12) = 1/2 & \text{if } u \in F \text{ and } u \neq 0 \\
sinu^*_F(u, -x) = -sinu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
sinu^*_F(-u, x) = -sinu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u
\end{array}$$

NOTE – $sinu^*_F(u, x) \approx x \cdot 2 \cdot \pi/u$ if $|x \cdot 2 \cdot \pi/u| < fminN_F$.

The $sinu_F$ operation:

$$sinu_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
sinu_F(u, x) & = result^*_F(sinu^*_F(u, x), nearest_F) & \\
& & \text{if } cycle_F(u, x) \in F \text{ and } cycle_F(u, x) \notin \{-u/2, 0, u/2\} \\
& = div_F(0, u) & \text{if } cycle_F(u, x) \in \{0, u/2\} \\
& = div_F(\mathbf{-0}, u) & \text{if } cycle_F(u, x) \in \{-u/2, \mathbf{-0}\} \\
& = cycle_F(u, x) & \text{otherwise}
\end{array}$$

5.3.9.3 Argument angular-unit cosine

The $cosu^*_F$ approximation helper function:

$$cosu^*_F : (u : F) \times F^u \to \mathcal{R}$$

$cosu^*_F(u, x)$ returns a close approximation to $\cos(x \cdot 2 \cdot \pi/u)$ in $\mathcal{R}$ if $u \neq 0$, with maximum error $max\_error\_sinu_F(u)$.

Further requirements on the $cosu^*_F$ approximation helper function are:

$$\begin{array}{ll}
cosu^*_F(u, n \cdot u + x) = cosu^*_F(u, x) & \text{if } n \in \mathcal{Z} \text{ and } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
cosu^*_F(u, 0) = 1 & \text{if } u \in F \text{ and } u \neq 0 \\
cosu^*_F(u, u/6) = 1/2 & \text{if } u \in F \text{ and } u \neq 0 \\
cosu^*_F(u, u/3) = -1/2 & \text{if } u \in F \text{ and } u \neq 0 \\
cosu^*_F(u, u/2) = -1 & \text{if } u \in F \text{ and } u \neq 0 \\
cosu^*_F(u, -x) = cosu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
cosu^*_F(-u, x) = cosu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u
\end{array}$$

NOTE – $cosu^*_F(u, x) = 1$ should hold if $|x \cdot 2 \cdot \pi/u| < \sqrt{epsilon_F/r_F}$

The $cosu_F$ operation:

$$cosu_F : F \times F \to F \cup \{\mathbf{underflow}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
cosu_F(u, x) & = result^*_F(cosu^*_F(u, x), nearest_F) & \\
& & \text{if } cycle_F(u, x) \in F \\
& = 1 & \text{if } cycle_F(u, x) = \mathbf{-0} \\
& = cycle_F(u, x) & \text{otherwise}
\end{array}$$

5.3.9.4 Argument angular-unit tangent

The $tanu^*_F$ approximation helper function:

$$tanu^*_F : (u : F) \times F^u \to \mathcal{R}$$

$tanu^*_F(u, x)$ returns a close approximation to $\tan(x \cdot 2 \cdot \pi/u)$ in $\mathcal{R}$ if $u \neq 0$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $tanu^*_F$ approximation helper function are:

$$\begin{array}{ll}
tanu^*_F(u, n \cdot u + x) = tanu^*_F(u, x) & \text{if } n \in \mathcal{Z} \text{ and } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
tanu^*_F(u, u/8) = 1 & \text{if } u \in F \text{ and } u \neq 0 \\
tanu^*_F(u, 3 \cdot u/8) = -1 & \text{if } u \in F \text{ and } u \neq 0 \\
tanu^*_F(u, -x) = -tanu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
tanu^*_F(-u, x) = -tanu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u
\end{array}$$

NOTE 1 – $tanu^*_F(u, x) \approx x \cdot 2 \cdot \pi/u$ if $|x \cdot 2 \cdot \pi/u| < fminN_F$.

The $tanu_F$ operation:

$$tanu_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
tanu_F(u, x) & = result^*_F(tanu^*_F(u, x), nearest_F) & \\
& & \text{if } cycle_F(u, x) \in F \text{ and} \\
& & \quad cycle_F(u, x) \notin \{-u/2, -u/4, 0, u/4, u/2\} \\
& = div_F(0, u) & \text{if } cycle_F(u, x) \in \{-u/2, 0\} \\
& = div_F(\mathbf{-0}, u) & \text{if } cycle_F(u, x) \in \{\mathbf{-0}, u/2\} \\
& = \mathbf{infinitary}(+\infty) & \text{if } cycle_F(u, x) = u/4 \\
& = \mathbf{infinitary}(-\infty) & \text{if } cycle_F(u, x) = -u/4 \\
& = cycle_F(u, x) & \text{otherwise}
\end{array}$$

NOTE 2 – The **infinitary** notification can arise for $tanu_F(u, x)$ only when $u/4$ is in F.

5.3.9.5 Argument angular-unit cotangent

The $cotu^*_F$ approximation helper function:

$$cotu^*_F : (u : F) \times F^u \to \mathcal{R}$$

$cotu^*_F(u, x)$ returns a close approximation to $\cot(x \cdot 2 \cdot \pi/u)$ in $\mathcal{R}$ if $u \neq 0$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $cotu^*_F$ approximation helper function are:

$$\begin{array}{ll}
cotu^*_F(u, n \cdot u + x) = cotu^*_F(u, x) & \text{if } n \in \mathcal{Z} \text{ and } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
cotu^*_F(u, u/8) = 1 & \text{if } u \in F \text{ and } u \neq 0 \\
cotu^*_F(u, 3 \cdot u/8) = -1 & \text{if } u \in F \text{ and } u \neq 0 \\
cotu^*_F(u, -x) = -cotu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
cotu^*_F(-u, x) = -cotu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u
\end{array}$$

The $cotu_F$ operation:

$$cotu_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
cotu_F(u, x) & = result^*_F(cotu^*_F(u, x), nearest_F) & \\
& & \text{if } cycle_F(u, x) \in F \text{ and} \\
& & \quad cycle_F(u, x) \notin \{-u/2, -u/4, 0, u/2\} \\
& = \mathbf{-0} & \text{if } cycle_F(u, x) = -u/4 \\
& = div_F(u, tanu_F(u, x)) & \text{if } cycle_F(u, x) \in \{-u/2, \mathbf{-0}, 0, u/2\} \\
& = cycle_F(u, x) & \text{otherwise}
\end{array}$$

5.3.9.6 Argument angular-unit secant

The $secu^*_F$ approximation helper function:

$$secu^*_F : (u : F) \times F^u \to \mathcal{R}$$

$secu^*_F(u, x)$ returns a close approximation to $\sec(x \cdot 2 \cdot \pi/u)$ in $\mathcal{R}$ if $u \neq 0$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $secu^*_F$ approximation helper function are:

$$\begin{array}{ll}
secu^*_F(u, n \cdot u + x) = secu^*_F(u, x) & \text{if } n \in \mathcal{Z} \text{ and } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
secu^*_F(u, 0) = 1 & \text{if } u \in F \text{ and } u \neq 0 \\
secu^*_F(u, u/6) = 2 & \text{if } u \in F \text{ and } u \neq 0 \\
secu^*_F(u, u/3) = -2 & \text{if } u \in F \text{ and } u \neq 0 \\
secu^*_F(u, u/2) = -1 & \text{if } u \in F \text{ and } u \neq 0 \\
secu^*_F(u, -x) = secu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
secu^*_F(-u, x) = secu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u
\end{array}$$

The $secu_F$ operation:

$$secu_F : F \times F \to F \cup \{\mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
secu_F(u, x) & = result^*_F(secu^*_F(u, x), nearest_F) & \\
& & \text{if } cycle_F(u, x) \in F \text{ and} \\
& & \quad cycle_F(u, x) \notin \{-u/4, u/4\} \\
& = div_F(1, cosu_F(u, x)) & \text{if } cycle_F(u, x) \in \{-u/4, \mathbf{-0}, u/4\} \\
& = cycle_F(u, x) & \text{otherwise}
\end{array}$$

5.3.9.7 Argument angular-unit cosecant

The $cscu^*_F$ approximation helper function:

$$cscu^*_F : (u : F) \times F^u \to \mathcal{R}$$

$cscu^*_F(u, x)$ returns a close approximation to $\csc(x \cdot 2 \cdot \pi/u)$ in $\mathcal{R}$ if $u \neq 0$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $cscu^*_F$ approximation helper function are:

$$\begin{array}{ll}
cscu^*_F(u, n \cdot u + x) = cscu^*_F(u, x) & \text{if } n \in \mathcal{Z} \text{ and } u \in F \text{ and } u \in 0 \text{ and } x \in F^u \\
cscu^*_F(u, u/12) = 2 & \text{if } u \in F \text{ and } u \neq 0 \\
cscu^*_F(u, u/4) = 1 & \text{if } u \in F \text{ and } u \neq 0 \\
cscu^*_F(u, 5 \cdot u/12) = 2 & \text{if } u \in F \text{ and } u \neq 0 \\
cscu^*_F(u, -x) = -cscu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u \\
cscu^*_F(-u, x) = -cscu^*_F(u, x) & \text{if } u \in F \text{ and } u \neq 0 \text{ and } x \in F^u
\end{array}$$

The $cscu_F$ operation:

$$cscu_F : F \times F \to F \cup \{\mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
cscu_F(u, x) & = result^*_F(cscu^*_F(u, x), nearest_F) & \\
& & \text{if } cycle_F(u, x) \in F \text{ and} \\
& & \quad cycle_F(u, x) \notin \{-u/2, 0, u/2\} \\
& = div_F(1, sinu_F(u, x)) & \text{if } cycle_F(u, x) \in \{-u/2, \mathbf{-0}, 0, u/2\} \\
& = cycle_F(u, x) & \text{otherwise}
\end{array}$$

5.3.9.8 Argument angular-unit cosine with sine

$$cossinu_F : F \times F \to (F \times (F \cup \{\mathbf{-0}\})) \cup \{\mathbf{underflow}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$cossinu_F(u,x) = (cosu_F(u,x), sinu_F(u,x))$$

NOTES

1 If there is an **absolute_precision_underflow** notification, then both result parts suffer from the **absolute_precision_underflow** and the continuation values for both parts are **qNaN**. Similarly for NaN and infinitary arguments, as well as an angular unit with too small absolute value.

2 If there is an **underflow** notification, only one of the result parts suffer from the underflow, and the other part has an absolute value greater than $fminN_F$.

5.3.9.9 Argument angular-unit arc sine

The $arcsinu^*_F$ approximation helper function:

$$arcsinu^*_F : F \times F \to \mathcal{R}$$

$arcsinu^*_F(u,x)$ returns a close approximation to $\arcsin(x) \cdot u/(2 \cdot \pi)$ in $\mathcal{R}$, with maximum error $max\_error\_sinu_F(u)$.

Further requirements on the $arcsinu^*_F$ approximation helper function are:

$$\begin{aligned}
&arcsinu^*_F(u, 1/2) = u/12 && \text{if } u \in F\\
&arcsinu^*_F(u, 1) = u/4 && \text{if } u \in F\\
&arcsinu^*_F(u, -x) = -arcsinu^*_F(u,x) && \text{if } u, x \in F\\
&arcsinu^*_F(-u, x) = -arcsinu^*_F(u,x) && \text{if } u, x \in F
\end{aligned}$$

NOTE – $arcsinu^*_F(u,x) \approx u/(2 \cdot \pi)$ if $|x| < fminN_F$.

The $arcsinu^\#_F$ range limitation helper function (for $u, x \in F$):

$$arcsinu^\#_F(u,x) = \max\{up_F(-|u/4|), \min\{arcsinu^*_F(u,x), down_F(|u/4|)\}\}$$

The $arcsinu_F$ operation:

$$arcsinu_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{invalid}\}$$

$$\begin{aligned}
arcsinu_F(u,x) &= result^*_F(arcsinu^\#_F(u,x), nearest_F)\\
&\qquad\qquad \text{if } u \in G_F \text{ and } x \in F \text{ and } |x| \leqslant 1 \text{ and } x \neq 0\\
&= mul_F(u,x) \qquad \text{if } u \in G_F \text{ and } x \in \{\mathbf{-0}, 0\}\\
&= no\_result2_F(u,x) \qquad \text{otherwise}
\end{aligned}$$

5.3.9.10 Argument angular-unit arc cosine

The $arccosu^*_F$ approximation helper function:

$$arccosu^*_F : F \times F \to \mathcal{R}$$

$arccosu^*_F(u,x)$ returns a close approximation to $\arccos(x) \cdot u/(2 \cdot \pi)$ in $\mathcal{R}$, with maximum error $max\_error\_sinu_F(u)$.

Further requirements on the $arccosu^*_F$ approximation helper function are:

$$\begin{aligned}
&arccosu^*_F(u, 1/2) = u/6 && \text{if } u \in F\\
&arccosu^*_F(u, 0) = u/4 && \text{if } u \in F\\
&arccosu^*_F(u, -1/2) = u/3 && \text{if } u \in F\\
&arccosu^*_F(u, -1) = u/2 && \text{if } u \in F\\
&arccosu^*_F(-u, x) = -arccosu^*_F(u, x) && \text{if } u, x \in F
\end{aligned}$$

The $arccosu^\#_F$ range limitation helper function (for $u, x \in F$):

$$arccosu^\#_F(u, x) = \max\{up_F(-|u/2|), \min\{arccosu^*_F(u, x), down_F(|u/2|)\}\}$$

The $arccosu_F$ operation:

$$arccosu_F : F \times F \to F \cup \{\textbf{underflow}, \textbf{invalid}\}$$

$$\begin{aligned}
arccosu_F(u, x) &= result^*_F(arccosu^\#_F(u, x), nearest_F)\\
&\qquad\qquad \text{if } u \in G_F \text{ and } x \in F \text{ and } |x| \leqslant 1\\
&= nearest_F(u/4) \qquad \text{if } u \in G_F \text{ and } x = \mathbf{-0}\\
&= no\_result2_F(u, x) \qquad \text{otherwise}
\end{aligned}$$

5.3.9.11 Argument angular-unit arc tangent

The $arctanu^*_F$ approximation helper function:

$$arctanu^*_F : F \times F \to \mathcal{R}$$

$arctanu^*_F(u, x)$ returns a close approximation to $\arctan(x) \cdot u/(2 \cdot \pi)$ in $\mathcal{R}$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $arctanu^*_F$ approximation helper function are:

$$\begin{aligned}
&arctanu^*_F(u, 1) = u/8 && \text{if } u \in F\\
&arctanu^*_F(u, x) = u/4 && \text{if } u, x \in F \text{ and } arctanu^*_F(u, x) \neq \arctan(x) \cdot u/(2 \cdot \pi)\\
& && \quad \text{and } x > 3 \cdot r_F/epsilon_F\\
&arctanu^*_F(u, -x) = -arctanu^*_F(u, x) && \text{if } u, x \in F\\
&arctanu^*_F(-u, x) = -arctanu^*_F(u, x) && \text{if } u, x \in F
\end{aligned}$$

NOTE 1 – $arctanu^*_F(u, x) \approx u/(2 \cdot \pi)$ if $|x| < fminN_F$

The $arctanu^\#_F$ range limitation helper function (for $u, x \in F$):

$$arctanu^\#_F(u, x) = \max\{up_F(-|u/4|), \min\{arctanu^*_F(u, x), down_F(|u/4|)\}\}$$

The $arctanu_F$ operation:

$$arctanu_F : F \times F \to F \cup \{\mathbf{-0}, \textbf{underflow}, \textbf{invalid}\}$$

$$\begin{aligned}
arctanu_F(u, x) &= result^*_F(arctanu^\#_F(u, x), nearest_F)\\
&\qquad\qquad \text{if } u \in G_F \text{ and } x \in F \text{ and } x \neq 0\\
&= mul_F(x, u) \qquad \text{if } u \in G_F \text{ and } x \in \{\mathbf{-0}, 0\}\\
&= up_F(-u/4) \qquad \text{if } u \in G_F \text{ and } x = \mathbf{-\infty} \text{ and } u > 0\\
&= down_F(u/4) \qquad \text{if } u \in G_F \text{ and } x = \mathbf{+\infty} \text{ and } u > 0\\
&= down_F(-u/4) \qquad \text{if } u \in G_F \text{ and } x = \mathbf{-\infty} \text{ and } u < 0\\
&= up_F(u/4) \qquad \text{if } u \in G_F \text{ and } x = \mathbf{+\infty} \text{ and } u < 0\\
&= no\_result2_F(u, x) \qquad \text{otherwise}
\end{aligned}$$

NOTE 2 – $arctanu_F(u, x) \approx arcu_F(u, 1, x)$. ($arcu_F$ is specified in subclause 5.3.9.15 below.)

5.3.9.12 Argument angular-unit arc cotangent

This clause specifies two inverse cotangent operations. One approximating the sign symmetric (but discontinuous at 0) arccot, the other approximating the continuous (but not sign symmetric) arccotc (both for non-radian angular units).

The $arccotu^*_F$ approximation helper function:

$$arccotu^*_F : F \times F \to \mathcal{R}$$

$arccotu^*_F(u, x)$ returns a close approximation to $\mathrm{arccot}(x) \cdot u/(2 \cdot \pi)$ in $\mathcal{R}$, with maximum error $max\_error\_tanu_F(u)$.

The $arccotcu^*_F$ approximation helper function:

$$arccotcu^*_F : F \times F \to \mathcal{R}$$

$arccotcu^*_F(u, x)$ returns a close approximation to $\mathrm{arccotc}(x) \cdot u/(2 \cdot \pi)$ in $\mathcal{R}$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $arccotu^*_F$ and $arccotcu^*_F$ approximation helper functions are:

$$\begin{array}{ll}
arccotu^*_F(u, 1) = u/8 & \text{if } u \in F \\
arccotu^*_F(u, 0) = u/4 & \text{if } u \in F \\
arccotu^*_F(u, -x) = -arccotu^*_F(u, x) & \text{if } u, x \in F \text{ and } x \neq 0 \\
 & \\
arccotcu^*_F(u, x) = arccotu^*_F(u, x) & \text{if } u, x \in F \text{ and } x \geqslant 0 \\
arccotcu^*_F(u, -1) = 3 \cdot u/8 & \text{if } u \in F \\
arccotcu^*_F(u, x) = u/2 & \text{if } u, x \in F \text{ and } arccotcu^*_F(u, x) \neq \mathrm{arccotc}(x) \cdot u/(2 \cdot \pi) \\
 & \quad \text{and } x < -3 \cdot r_F/epsilon_F \\
arccotcu^*_F(-u, x) = -arccotcu^*_F(u, x) & \text{if } u, x \in F
\end{array}$$

The $arccotu^{\#}_F$ and $arccotcu^{\#}_F$ range limitation helper functions (for $u, x \in F$):

$$arccotu^{\#}_F(u, x) = \max\{up_F(-|u/4|), \min\{arccotu^*_F(u, x), down_F(|u/4|)\}\}$$
$$arccotcu^{\#}_F(u, x) = \max\{up_F(-|u/2|), \min\{arccotcu^*_F(u, x), down_F(|u/2|)\}\}$$

The $arccotu_F$ operation:

$$arccotu_F : F \times F \to F \cup \{\mathbf{underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
arccotu_F(u, x) & = result^*_F(arccotu^{\#}_F(u, x), nearest_F) & \\
 & & \text{if } u \in G_F \text{ and } x \in F \\
 & = neg_F(arccotu_F(u, 0)) & \text{if } u \in G_F \text{ and } x = \mathbf{-0} \\
 & = div_F(u, x) & \text{if } u \in G_F \text{ and } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
 & = no\_result2_F(u, x) & \text{otherwise}
\end{array}$$

NOTES

1 $arccotu_F(u, neg_F(x)) = neg_F(arccotu_F(u, x))$.

2 Due to the range limitation, $arccotu_F(u, 0)$ need not equal $arccotcu_F(u, 0)$.

The $arccotcu_F$ operation:

$$arccotcu_F : F \times F \to F \cup \{\mathbf{underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
arccotcu_F(u, x) & = result^*_F(arccotcu^{\#}_F(u, x), nearest_F) & \\
 & & \text{if } u \in G_F \text{ and } x \in F \\
 & = nearest_F(u/4) & \text{if } u \in G_F \text{ and } x = \mathbf{-0} \\
 & = down_F(u/2) & \text{if } u \in G_F \text{ and } x = \mathbf{-\infty} \text{ and } u > 0
\end{array}$$

$$\begin{array}{ll} = up_F(u/2) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } u < 0 \\ = div_F(u,x) & \text{if } u \in G_F \text{ and } x = +\infty \\ = no\_result2_F(u,x) & \text{otherwise} \end{array}$$

NOTE 3 – $arccotcu_F(u,x) \approx arcu_F(u,x,1)$. ($arcu_F$ is specified in subclause 5.3.9.15 below.)

5.3.9.13 Argument angular-unit arc secant

The $arcsecu^*_F$ approximation helper function:

$$arcsecu^*_F : F \times F \to \mathcal{R}$$

$arcsecu^*_F(u,x)$ returns a close approximation to $\text{arcsec}(x) \cdot u/(2 \cdot \pi)$ in $\mathcal{R}$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $arcsecu^*_F$ approximation helper function are:

$$\begin{array}{ll} arcsecu^*_F(u,2) = u/6 & \text{if } u \in F \\ arcsecu^*_F(u,-2) = u/3 & \text{if } u \in F \\ arcsecu^*_F(u,-1) = u/2 & \text{if } u \in F \\ arcsecu^*_F(u,x) \leqslant u/4 & \text{if } u,x \in F \text{ and } x > 0 \text{ and } u > 0 \\ arcsecu^*_F(u,x) \geqslant u/4 & \text{if } u,x \in F \text{ and } x < 0 \text{ and } u > 0 \\ arcsecu^*_F(u,x) = u/4 & \text{if } u,x \in F \text{ and } arcsecu^*_F(u,x) \neq \text{arcsec}(x) \cdot u/(2 \cdot \pi) \\ & \quad \text{and } |x| > 3 \cdot r_F/epsilon_F \\ arcsecu^*_F(-u,x) = -arcsecu^*_F(u,x) & \text{if } u,x \in F \end{array}$$

The $arcsecu^{\#}_F$ range limitation helper function (for $u,x \in F$):

$$\begin{array}{lll} arcsecu^{\#}_F(u,x) & = \max\{up_F(-|u/4|), \min\{arcsecu^*_F(u,x), down_F(|u/4|)\}\} & \\ & & \text{if } x \geqslant 1 \\ & = \max\{up_F(u/4), \min\{arcsecu^*_F(u,x), down_F(u/2)\}\} & \\ & & \text{if } x \leqslant -1 \text{ and } u > 0 \\ & = \max\{up_F(u/2), \min\{arcsecu^*_F(u,x), down_F(u/4)\}\} & \\ & & \text{if } x \leqslant -1 \text{ and } u < 0 \end{array}$$

The $arcsecu_F$ operation:

$$arcsecu_F : F \times F \to F \cup \{\mathbf{underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll} arcsecu_F(u,x) & = result^*_F(arcsecu^{\#}_F(u,x), nearest_F) & \\ & & \text{if } u \in G_F \text{ and } x \in F \text{ and } 1 \leqslant |x| \\ & = down_F(u/4) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } u > 0 \\ & = up_F(u/4) & \text{if } u \in G_F \text{ and } x = +\infty \text{ and } u > 0 \\ & = up_F(u/4) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } u < 0 \\ & = down_F(u/4) & \text{if } u \in G_F \text{ and } x = +\infty \text{ and } u < 0 \\ & = no\_result2_F(u,x) & \text{otherwise} \end{array}$$

5.3.9.14 Argument angular-unit arc cosecant

The $arccscu^*_F$ approximation helper function:

$$arccscu^*_F : F \times F \to \mathcal{R}$$

$arccscu^*_F(u,x)$ returns a close approximation to $\mathrm{arccsc}(x) \cdot u/(2 \cdot \pi)$ in $\mathcal{R}$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $arccscu^*_F$ approximation helper function are:

$$\begin{array}{ll}
arccscu^*_F(u,2) = u/12 & \text{if } u \in F \\
arccscu^*_F(u,1) = u/4 & \text{if } u \in F \\
arccscu^*_F(u,-x) = -arccscu^*_F(u,x) & \text{if } u,x \in F \\
arccscu^*_F(-u,x) = -arccscu^*_F(u,x) & \text{if } u,x \in F
\end{array}$$

The $arccscu^\#_F$ range limitation helper function (for $u, x \in F$):

$$arccscu^\#_F(u,x) = \max\{up_F(-|u/4|), \min\{arccscu^*_F(u,x), down_F(|u/4|)\}\}$$

The $arccscu_F$ operation:

$$arccscu_F : F \times F \to F \cup \{\mathbf{underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
arccscu_F(u,x) & = result^*_F(arccscu^\#_F(u,x), nearest_F) & \\
 & & \text{if } u \in G_F \text{ and } x \in F \text{ and } 1 \leqslant |x| \\
 & = mul_F(-u,0) & \text{if } u \in G_F \text{ and } x = \mathbf{-\infty} \\
 & = mul_F(u,0) & \text{if } u \in G_F \text{ and } x = \mathbf{+\infty} \\
 & = no\_result2_F(u,x) & \text{otherwise}
\end{array}$$

5.3.9.15 Argument angular-unit angle from Cartesian co-ordinates

The $arcu^*_F$ approximation helper function:

$$arcu^*_F : F \times F \times F \to \mathcal{R}$$

$arcu^*_F(u,x,y)$ returns a close approximation to $arc(x,y) \cdot u/(2 \cdot \pi)$ in $\mathcal{R}$, with maximum error $max\_error\_tanu_F(u)$.

Further requirements on the $arcu^*_F$ approximation helper function are:

$$\begin{array}{ll}
arcu^*_F(u,x,x) = u/8 & \text{if } u,x \in F \text{ and } x > 0 \\
arcu^*_F(u,0,y) = u/4 & \text{if } u,y \in F \text{ and } y > 0 \\
arcu^*_F(u,x,-x) = 3 \cdot u/8 & \text{if } u,x \in F \text{ and } x < 0 \\
arcu^*_F(u,x,0) = u/2 & \text{if } u,x \in F \text{ and } x < 0 \\
arcu^*_F(u,x,-y) = -arcu^*_F(u,x,y) & \text{if } u,x,y \in F \text{ and } (y \neq 0 \text{ or } x > 0) \\
arcu^*_F(-u,x,y) = -arcu^*_F(u,x,y) & \text{if } u,x,y \in F
\end{array}$$

The $arcu^\#_F$ range limitation helper function (for $u, x, y \in F$):

$$arcu^\#_F(u,x,y) = \max\{up_F(-|u/2|), \min\{arcu^*_F(u,x,y), down_F(|u/2|)\}\}$$

The $arcu_F$ operation:

$$arcu_F : F \times F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
arcu_F(u,x,y) & = result^*_F(arcu^\#_F(u,x,y), nearest_F) & \\
 & & \text{if } u \in G_F \text{ and } x,y \in F \text{ and } (x < 0 \text{ or } y \neq 0) \\
 & = mul_F(u,0) & \text{if } u \in G_F \text{ and } x \in F \text{ and } x \geqslant 0 \text{ and } y = 0 \\
 & = down_F(u/2) & \text{if } u \in G_F \text{ and } x = \mathbf{-0} \text{ and } y = 0 \text{ and } u > 0 \\
 & = up_F(u/2) & \text{if } u \in G_F \text{ and } x = \mathbf{-0} \text{ and } y = 0 \text{ and } u < 0 \\
 & = arcu_F(u,0,y) & \text{if } u \in G_F \text{ and } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and} \\
 & & \quad y \neq 0 \\
 & = neg_F(arcu_F(u,x,0)) & \text{if } u \in G_F \text{ and } y = \mathbf{-0} \text{ and } x \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\}
\end{array}$$

$$
\begin{array}{ll}
= mul_F(0, u) & \text{if } u \in G_F \text{ and } x = +\infty \text{ and } y \in F \text{ and } y \geqslant 0 \\
= mul_F(0, -u) & \text{if } u \in G_F \text{ and } x = +\infty \text{ and } y \in F \text{ and } y < 0 \\
= nearest_F(u/8) & \text{if } u \in G_F \text{ and } x = +\infty \text{ and } y = +\infty \\
= nearest_F(u/4) & \text{if } u \in G_F \text{ and } x \in F \text{ and } y = +\infty \\
= nearest_F(3 \cdot u/8) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } y = +\infty \\
= down_F(u/2) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } y \in F \text{ and} \\
 & \quad y \geqslant 0 \text{ and } u > 0 \\
= up_F(-u/2) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } y \in F \text{ and} \\
 & \quad y < 0 \text{ and } u > 0 \\
= up_F(u/2) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } y \in F \text{ and} \\
 & \quad y > 0 \text{ and } u < 0 \\
= down_F(-u/2) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } y \in F \text{ and} \\
 & \quad y \leqslant 0 \text{ and } u < 0 \\
= nearest_F(-3 \cdot u/8) & \text{if } u \in G_F \text{ and } x = -\infty \text{ and } y = -\infty \\
= nearest_F(-u/4) & \text{if } u \in G_F \text{ and } x \in F \text{ and } y = -\infty \\
= nearest_F(-u/8) & \text{if } u \in G_F \text{ and } x = +\infty \text{ and } y = -\infty \\
 & \\
= no\_result3_F(u, x, y) & \text{otherwise}
\end{array}
$$

NOTE – Note that the arc operations do *not* return an **invalid** notification at the origin (both second and third arguments in $\{-\mathbf{0}, 0\}$). See B.5.3.8 and B.5.3.9. Bindings may choose to alter this behaviour.

5.3.10 Operations for angular-unit conversions

5.3.10.1 Converting radian angle to argument angular-unit angle

Define the mathematical function:

$$rad\_to\_cycle : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$$

$$
\begin{array}{lll}
rad\_to\_cycle(x, w) & = \arccos(\cos(x)) \cdot w/(2 \cdot \pi) & \text{if } \sin(x) \geqslant 0 \text{ and } w \neq 0 \\
 & = -\arccos(\cos(x)) \cdot w/(2 \cdot \pi) & \text{if } \sin(x) < 0 \text{ and } w \neq 0
\end{array}
$$

The $rad\_to\_cycle^*_F$ approximation helper function:

$$rad\_to\_cycle^*_F : F^{2 \cdot \pi} \times F \to \mathcal{R}$$

$rad\_to\_cycle^*_F(x, w)$ returns a close approximation to $rad\_to\_cycle(x, w)$ in $\mathcal{R}$, with maximum error $max\_error\_rad_F$, if $|x| \leqslant big\_angle\_r_F$.

Further requirements on the $rad\_to\_cycle^*_F$ approximation helper function are (for $w \in F$):

$$
\begin{array}{ll}
rad\_to\_cycle^*_F(n \cdot 2 \cdot \pi + \pi/6, w) = w/12 & \text{if } n \in \mathcal{Z} \text{ and } |n \cdot 2 \cdot \pi + \pi/6| \leqslant big\_angle\_r_F \\
rad\_to\_cycle^*_F(n \cdot 2 \cdot \pi + \pi/4, w) = w/8 & \text{if } n \in \mathcal{Z} \text{ and } |n \cdot 2 \cdot \pi + \pi/4| \leqslant big\_angle\_r_F \\
rad\_to\_cycle^*_F(n \cdot 2 \cdot \pi + \pi/3, w) = w/6 & \text{if } n \in \mathcal{Z} \text{ and } |n \cdot 2 \cdot \pi + \pi/3| \leqslant big\_angle\_r_F \\
rad\_to\_cycle^*_F(n \cdot 2 \cdot \pi + \pi/2, w) = w/4 & \text{if } n \in \mathcal{Z} \text{ and } |n \cdot 2 \cdot \pi + \pi/2| \leqslant big\_angle\_r_F \\
rad\_to\_cycle^*_F(n \cdot 2 \cdot \pi + 2 \cdot \pi/3, w) = w/3 & \\
 & \text{if } n \in \mathcal{Z} \text{ and } |n \cdot 2 \cdot \pi + 2 \cdot \pi/3| \leqslant big\_angle\_r_F \\
rad\_to\_cycle^*_F(n \cdot 2 \cdot \pi + 3 \cdot \pi/4, w) = 3 \cdot w/8 & \\
 & \text{if } n \in \mathcal{Z} \text{ and } |n \cdot 2 \cdot \pi + 3 \cdot \pi/4| \leqslant big\_angle\_r_F
\end{array}
$$

$rad\_to\_cycle^*_F(n \cdot 2 \cdot \pi + 5 \cdot \pi/6, w) = 5 \cdot w/12$
  if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi + 5 \cdot \pi/6| \leqslant big\_angle\_r_F$
$rad\_to\_cycle^*_F(n \cdot 2 \cdot \pi + \pi, w) = w/2$ if $n \in \mathcal{Z}$ and $|n \cdot 2 \cdot \pi + \pi| \leqslant big\_angle\_r_F$
$rad\_to\_cycle^*_F(-x, w) = -rad\_to\_cycle^*_F(x, w)$
  if $x \in F^{2 \cdot \pi}$ and $rad\_to\_cycle(x, w) \neq w/2$
$rad\_to\_cycle^*_F(x, -w) = -rad\_to\_cycle^*_F(x, w)$
  if $x \in F^{2 \cdot \pi}$ and $rad\_to\_cycle(x, w) \neq w/2$

The $rad\_to\_cycle_F$ operation:

$rad\_to\_cycle_F : F \times F \to F \cup \{\textbf{underflow}, \textbf{absolute\_precision\_underflow}, \textbf{invalid}\}$

$rad\_to\_cycle_F(x, w)$
$= result^*_F(rad\_to\_cycle^*_F(x, w), nearest_F)$
  if $w \in G_F$ and $x \in F$ and $|x| \leqslant big\_angle\_r_F$ and $x \neq 0$
$= mul_F(w, x)$ if $w \in G_F$ and $x \in \{\mathbf{-0}, 0\}$
$= \textbf{absolute\_precision\_underflow}(\textbf{qNaN})$
  if $w \in G_F$ and $x \in F$ and $|x| > big\_angle\_r_F$
$= no\_result2_F(x, w)$ otherwise

5.3.10.2 Converting argument angular-unit angle to radian angle

Define the mathematical function:

$cycle\_to\_rad : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$

$$\begin{aligned} cycle\_to\_rad(u, x) &= \arccos(\cos(x \cdot 2 \cdot \pi/u)) && \text{if } \sin(x \cdot 2 \cdot \pi/u) \geqslant 0 \\ &= -\arccos(\cos(x \cdot 2 \cdot \pi/u)) && \text{if } \sin(x \cdot 2 \cdot \pi/u) < 0 \end{aligned}$$

The $cycle\_to\_rad^*_F$ approximation helper function:

$cycle\_to\_rad^*_F : (u : F) \times F^u \to \mathcal{R}$

$cycle\_to\_rad^*_F(u, x)$ returns a close approximation to $cycle\_to\_rad(u, x)$ in $\mathcal{R}$, if $u \neq 0$, with maximum error $max\_error\_rad_F$.

Further requirements on the $cycle\_to\_rad^*_F$ approximation helper function are (for $u \in F$, $u \neq 0$):

$cycle\_to\_rad^*_F(u, n \cdot u + x) = cycle\_to\_rad^*_F(u, x)$
  if $n \in \mathcal{Z}$ and $x \in F^u$
$cycle\_to\_rad^*_F(u, u/12) = \pi/6$
$cycle\_to\_rad^*_F(u, u/8) = \pi/4$
$cycle\_to\_rad^*_F(u, u/6) = \pi/3$
$cycle\_to\_rad^*_F(u, u/4) = \pi/2$
$cycle\_to\_rad^*_F(u, u/3) = 2 \cdot \pi/3$
$cycle\_to\_rad^*_F(u, 3 \cdot u/8) = 3 \cdot \pi/4$
$cycle\_to\_rad^*_F(u, 5 \cdot u/12) = 5 \cdot \pi/6$
$cycle\_to\_rad^*_F(u, u/2) = \pi$
$cycle\_to\_rad^*_F(u, -x) = -cycle\_to\_rad^*_F(u, x)$
  if $x \in F^u$ and $cycle\_to\_rad(u, x) \neq \pi$
$cycle\_to\_rad^*_F(-u, x) = -cycle\_to\_rad^*_F(u, x)$
  if $x \in F^u$ and $cycle\_to\_rad(u, x) \neq \pi$

The $cycle\_to\_rad_F$ operation:

$$cycle\_to\_rad_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{aligned} cycle\_to\_rad_F(u, x) &= result^*_F(cycle\_to\_rad^*_F(u, x), nearest_F) \\ &\qquad \text{if } cycle_F(u, x) \in F \text{ and } cycle_F(u, x) \neq 0 \\ &= mul_F(cycle_F(u, x), u) \quad \text{if } cycle_F(u, x) \in \{\mathbf{-0}, 0\} \\ &= cycle_F(u, x) \quad \text{otherwise} \end{aligned}$$

5.3.10.3 Converting argument angular-unit angle to (another) argument angular-unit angle

Define the mathematical function:

$$cycle\_to\_cycle : \mathcal{R} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R}$$

$$\begin{aligned} cycle\_to\_cycle(u, x, w) &= \arccos(\cos(x \cdot 2 \cdot \pi/u)) \cdot w/(2 \cdot \pi) \\ &\qquad \text{if } u \neq 0 \text{ and } w \neq 0 \text{ and } \sin(x \cdot 2 \cdot \pi/u) \geqslant 0 \\ &= -\arccos(\cos(x \cdot 2 \cdot \pi/u)) \cdot w/(2 \cdot \pi) \\ &\qquad \text{if } u \neq 0 \text{ and } w \neq 0 \text{ and } \sin(x \cdot 2 \cdot \pi/u) < 0 \end{aligned}$$

The $cycle\_to\_cycle^*_F$ approximation helper function:

$$cycle\_to\_cycle^*_F : (u : F) \times F^u \times F \to \mathcal{R}$$

$cycle\_to\_cycle^*_F(u, x, w)$ returns a close approximation to $cycle\_to\_cycle(u, x, w)$ in $\mathcal{R}$ if $u \neq 0$ and $|x/u| \leqslant big\_angle\_u_F$, with maximum error $max\_error\_rad_F$.

Further requirements on the $cycle\_to\_cycle^*_F$ approximation helper function are (for $u, w \in F$, $u \neq 0$):

$$\begin{aligned} &cycle\_to\_cycle^*_F(u, n \cdot u + x, w) = cycle\_to\_cycle^*_F(u, x, w) \\ &\qquad \text{if } n \in \mathcal{Z} \text{ and } x \in F^u \\ &cycle\_to\_cycle^*_F(u, u/12, w) = w/12 \\ &cycle\_to\_cycle^*_F(u, u/8, w) = w/8 \\ &cycle\_to\_cycle^*_F(u, u/6, w) = w/6 \\ &cycle\_to\_cycle^*_F(u, u/4, w) = w/4 \\ &cycle\_to\_cycle^*_F(u, u/3, w) = w/3 \\ &cycle\_to\_cycle^*_F(u, 3 \cdot u/8, w) = 3 \cdot w/8 \\ &cycle\_to\_cycle^*_F(u, 5 \cdot u/12, w) = 5 \cdot w/12 \\ &cycle\_to\_cycle^*_F(u, u/2, w) = w/2 \\ &cycle\_to\_cycle^*_F(u, -x, w) = -cycle\_to\_cycle^*_F(u, x, w) \\ &\qquad \text{if } x \in F^u \text{ and } cycle\_to\_cycle(u, x, w) \neq w/2 \\ &cycle\_to\_cycle^*_F(-u, x, w) = -cycle\_to\_cycle^*_F(u, x, w) \\ &\qquad \text{if } x \in F^u \text{ and } cycle\_to\_cycle(u, x, w) \neq w/2 \\ &cycle\_to\_cycle^*_F(u, x, -w) = -cycle\_to\_cycle^*_F(u, x, w) \\ &\qquad \text{if } x \in F^u \text{ and } cycle\_to\_cycle(u, x, w) \neq w/2 \end{aligned}$$

The $cycle\_to\_cycle_F$ operation:

$$cycle\_to\_cycle_F : F \times F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{absolute\_precision\_underflow}, \mathbf{invalid}\}$$

$$\begin{aligned}
&\mathit{cycle\_to\_cycle}_F(u,x,w)\\
&\quad = \mathit{result}^*_F(\mathit{cycle\_to\_cycle}^*_F(u,x,w), \mathit{nearest}_F)\\
&\qquad\qquad \text{if } w \in G_F \text{ and } \mathit{cycle}_F(u,x) \in F \text{ and } \mathit{cycle}_F(u,x) \neq 0\\
&\quad = \mathit{mul}_F(w, \mathit{cycle}_F(u,x)) \quad \text{if } w \in G_F \text{ and } \mathit{cycle}_F(u,x) \in \{\mathbf{-0}, 0\}\\
&\quad = \mathbf{absolute\_precision\_underflow}(\mathbf{qNaN})\\
&\qquad\qquad \text{if } w \in G_F \text{ and}\\
&\qquad\qquad\quad \mathit{cycle}_F(u,x) = \mathbf{absolute\_precision\_underflow}\\
&\quad = \mathit{no\_result3}_F(u,x,w) \quad \text{otherwise}
\end{aligned}$$

5.3.11 Operations for hyperbolic elementary functions

There shall be two maximum error parameters for operations corresponding to the hyperbolic and inverse hyperbolic functions:

$\mathit{max\_error\_sinh}_F \in F$

$\mathit{max\_error\_tanh}_F \in F$

The $\mathit{max\_error\_sinh}_F$ parameter shall have a value that is $\leqslant 2 \cdot \mathit{rnd\_error}_F$. The $\mathit{max\_error\_tanh}_F$ parameter shall have a value that is $\leqslant 2 \cdot \mathit{rnd\_error}_F$.

5.3.11.1 Hyperbolic sine

The sinh^*_F approximation helper function:

$\mathit{sinh}^*_F : F \to \mathcal{R}$

$\mathit{sinh}^*_F(x)$ returns a close approximation to $\sinh(x)$ in $\mathcal{R}$, with maximum error $\mathit{max\_error\_sinh}_F$.

Further requirements on the sinh^*_F approximation helper function are:

$$\begin{aligned}
&\mathit{sinh}^*_F(x) = x && \text{if } x \in F \text{ and } \mathit{sinh}^*_F(x) \neq \sinh(x) \text{ and}\\
& && \quad |x| < \sqrt{2 \cdot \mathit{epsilon}_F / r_F}\\
&\mathit{sinh}^*_F(-x) = -\mathit{sinh}^*_F(x) && \text{if } x \in F
\end{aligned}$$

The sinh_F operation:

$\mathit{sinh}_F : F \to F \cup \{\mathbf{overflow}\}$

$$\begin{aligned}
\mathit{sinh}_F(x) &= \mathit{result}^*_F(\mathit{sinh}^*_F(x), \mathit{nearest}_F)\\
&\qquad\qquad \text{if } x \in F \text{ and } |x| > \mathit{fminN}_F\\
&= x \qquad\qquad \text{if } x \in F \text{ and } |x| \leqslant \mathit{fminN}_F\\
&= x \qquad\qquad \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\}\\
&= \mathit{no\_result}_F(x) \qquad \text{otherwise}
\end{aligned}$$

NOTES

1 **underflow** is explicitly avoided.

2 $\mathit{sinh}_F(x)$ will overflow approximately when $|x| > \ln(2 \cdot \mathit{fmax}_F)$.

5.3.11.2 Hyperbolic cosine

The cosh^*_F approximation helper function:

$\mathit{cosh}^*_F : F \to \mathcal{R}$

$cosh^*_F(x)$ returns a close approximation to $\cosh(x)$ in $\mathcal{R}$, with maximum error $max\_error\_sinh_F$.

Further requirements on the $cosh^*_F$ approximation helper function are:

$cosh^*_F(x) = 1$ if $x \in F$ and $cosh^*_F(x) \neq \cosh(x)$ and $|x| < \sqrt{epsilon_F}$

$cosh^*_F(-x) = cosh^*_F(x)$ if $x \in F$

The relationship to the $sinh^*_F$ approximation helper function for the $sinh_F$ operation in the same library shall be:

$cosh^*_F(x) \geqslant sinh^*_F(x)$ if $x \in F$

The $cosh_F$ operation:

$cosh_F : F \to F \cup \{\textbf{overflow}\}$

$$
\begin{array}{lll}
cosh_F(x) & = result^*_F(cosh^*_F(x), nearest_F) & \\
 & & \text{if } x \in F \\
 & = 1 & \text{if } x = \mathbf{-0} \\
 & = \mathbf{+\infty} & \text{if } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
 & = no\_result_F(x) & \text{otherwise}
\end{array}
$$

NOTE – $cosh_F(x)$ will overflow approximately when $|x| > \ln(2 \cdot fmax_F)$.

5.3.11.3 Hyperbolic tangent

The $tanh^*_F$ approximation helper function:

$tanh^*_F : F \to \mathcal{R}$

$tanh^*_F(x)$ returns a close approximation to $\tanh(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tanh_F$.

Further requirements on the $tanh^*_F$ approximation helper function are:

$tanh^*_F(x) = x$ if $x \in F$ and $tanh^*_F(x) \neq \tanh(x)$ and $|x| \leqslant \sqrt{1.5 \cdot epsilon_F / r_F}$

$tanh^*_F(x) = 1$ if $x \in F$ and $tanh^*_F(x) \neq \tanh(x)$ and $x > \operatorname{arctanh}(1 - (epsilon_F/(3 \cdot r_F)))$

$tanh^*_F(-x) = -tanh^*_F(x)$ if $x \in F$

The $tanh_F$ operation:

$tanh_F : F \to F$

$$
\begin{array}{lll}
tanh_F(x) & = result^*_F(tanh^*_F(x), nearest_F) & \\
 & & \text{if } x \in F \text{ and } |x| > fminN_F \\
 & = x & \text{if } x \in F \text{ and } |x| \leqslant fminN_F \\
 & = \mathbf{-0} & \text{if } x = \mathbf{-0} \\
 & = -1 & \text{if } x = \mathbf{-\infty} \\
 & = 1 & \text{if } x = \mathbf{+\infty} \\
 & = no\_result_F(x) & \text{otherwise}
\end{array}
$$

NOTE – **underflow** is explicitly avoided.

5.3.11.4 Hyperbolic cotangent

The $coth^*_F$ approximation helper function:

$$coth^*_F : F \to \mathcal{R}$$

$coth^*_F(x)$ returns a close approximation to $\coth(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tanh_F$.

Further requirements on the $coth^*_F$ approximation helper function are:

$coth^*_F(x) = 1$	if $x \in F$ and $coth^*_F(x) \neq \coth(x)$ and $x > \text{arccoth}(1 + (epsilon_F/4))$
$coth^*_F(-x) = -coth^*_F(x)$	if $x \in F$

The $coth_F$ operation:

$$coth_F : F \to F \cup \{\textbf{infinitary}, \textbf{overflow}\}$$

$$\begin{array}{lll} coth_F(x) & = result^*_F(coth^*_F(x), nearest_F) & \\ & & \text{if } x \in F \text{ and } x \neq 0 \\ & = \textbf{infinitary}(+\infty) & \text{if } x = 0 \\ & = \textbf{infinitary}(-\infty) & \text{if } x = \mathbf{-0} \\ & = -1 & \text{if } x = \mathbf{-\infty} \\ & = 1 & \text{if } x = \mathbf{+\infty} \\ & = no\_result_F(x) & \text{otherwise} \end{array}$$

NOTE – $coth_F(x)$ will overflow approximately when $|1/x| > fmax_F$.

5.3.11.5 Hyperbolic secant

The $sech^*_F$ approximation helper function:

$$sech^*_F : F \to \mathcal{R}$$

$sech^*_F(x)$ returns a close approximation to $\text{sech}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tanh_F$.

Further requirements on the $sech^*_F$ approximation helper function are:

$sech^*_F(x) = 1$	if $x \in F$ and $sech^*_F(x) \neq \text{sech}(x)$ and $\lvert x\rvert < \sqrt{epsilon_F/r_F}$
$sech^*_F(-x) = sech^*_F(x)$	if $x \in F$
$sech^*_F(x) < fminD_F/2$	if $x \in F$ and $x > 2 - \ln(fminD_F/4)$

The $sech_F$ operation:

$$sech_F : F \to F \cup \{\textbf{underflow}\}$$

$$\begin{array}{lll} sech_F(x) & = result^*_F(sech^*_F(x), nearest_F) & \\ & & \text{if } x \in F \\ & = 1 & \text{if } x = \mathbf{-0} \\ & = 0 & \text{if } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\ & = no\_result_F(x) & \text{otherwise} \end{array}$$

5.3.11.6 Hyperbolic cosecant

The $csch^*_F$ approximation helper function:

$csch^*_F : F \to \mathcal{R}$

$csch^*_F(x)$ returns a close approximation to $\operatorname{csch}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tanh_F$.

Further requirements on the $csch^*_F$ approximation helper function are:

$$\begin{array}{ll} csch^*_F(-x) = -csch^*_F(x) & \text{if } x \in F \\ csch^*_F(x) < fminD_F/2 & \text{if } x \in F \text{ and } x > 2 - \ln(fminD_F/4) \end{array}$$

The relationship to the $sech^*_F$ approximation helper function for the $sech_F$ operation in the same library shall be:

$$\begin{array}{ll} csch^*_F(x) \geqslant sech^*_F(x) & \text{if } x \in F \text{ and } x > 0 \end{array}$$

The $csch_F$ operation:

$csch_F : F \to F \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}\}$

$$\begin{array}{lll} csch_F(x) & = result^*_F(csch^*_F(x), nearest_F) & \\ & & \text{if } x \in F \text{ and } x \neq 0 \\ & = div_F(1, x) & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, 0, \mathbf{+\infty}\} \\ & = no\_result_F(x) & \text{otherwise} \end{array}$$

NOTE – $csch_F(x)$ will overflow approximately when $|1/x| > fmax_F$.

5.3.11.7 Inverse hyperbolic sine

The $arcsinh^*_F$ approximation helper function:

$arcsinh^*_F : F \to \mathcal{R}$

$arcsinh^*_F(x)$ returns a close approximation to $\operatorname{arcsinh}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_sinh_F$.

Further requirements on the $arcsinh^*_F$ approximation helper function are:

$$\begin{array}{ll} arcsinh^*_F(x) = x & \text{if } x \in F \text{ and } arcsinh^*_F(x) \neq \operatorname{arcsinh}(x) \text{ and} \\ & \quad |x| \leqslant \sqrt{3 \cdot epsilon_F / r_F} \\ arcsinh^*_F(-x) = -arcsinh^*_F(x) & \text{if } x \in F \end{array}$$

The $arcsinh_F$ operation:

$arcsinh_F : F \to F$

$$\begin{array}{lll} arcsinh_F(x) & = result^*_F(arcsinh^*_F(x), nearest_F) & \\ & & \text{if } x \in F \text{ and } |x| > fminN_F \\ & = x & \text{if } x \in F \text{ and } |x| \leqslant fminN_F \\ & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = no\_result_F(x) & \text{otherwise} \end{array}$$

NOTE – **underflow** is explicitly avoided.

5.3.11.8 Inverse hyperbolic cosine

The $arccosh^*_F$ approximation helper function:

$arccosh^*_F : F \to \mathcal{R}$

$arccosh^*_F(x)$ returns a close approximation to $\mathrm{arccosh}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_sinh_F$.

The relationship to the $arcsinh^*_F$ approximation helper function for the $arcsinh_F$ operation in the same library shall be:

$arccosh^*_F(x) \leqslant arcsinh^*_F(x)$ if $x \in F$

The $arccosh_F$ operation:

$arccosh_F : F \to F \cup \{\mathbf{invalid}\}$

$$\begin{aligned}
arccosh_F(x) &= result^*_F(arccosh^*_F(x), nearest_F) && \text{if } x \in F \text{ and } x \geqslant 1\\
&= \mathbf{+\infty} && \text{if } x = \mathbf{+\infty}\\
&= no\_result_F(x) && \text{otherwise}
\end{aligned}$$

5.3.11.9 Inverse hyperbolic tangent

The $arctanh^*_F$ approximation helper function:

$arctanh^*_F : F \to \mathcal{R}$

$arctanh^*_F(x)$ returns a close approximation to $\mathrm{arctanh}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tanh_F$.

Further requirements on the $arctanh^*_F$ approximation helper function are:

$$\begin{aligned}
arctanh^*_F(x) &= x && \text{if } x \in F \text{ and } arctanh^*_F(x) \neq \mathrm{arctanh}(x) \text{ and } |x| < \sqrt{epsilon_F/r_F}\\
arctanh^*_F(-x) &= -arctanh^*_F(x) && \text{if } x \in F
\end{aligned}$$

The $arctanh_F$ operation:

$arctanh_F : F \to F \cup \{\mathbf{infinitary}, \mathbf{invalid}\}$

$$\begin{aligned}
arctanh_F(x) &= result^*_F(arctanh^*_F(x), nearest_F) && \text{if } x \in F \text{ and } fminN_F < |x| < 1\\
&= x && \text{if } x \in F \text{ and } |x| \leqslant fminN_F\\
&= \mathbf{-0} && \text{if } x = \mathbf{-0}\\
&= \mathbf{infinitary}(\mathbf{+\infty}) && \text{if } x = 1\\
&= \mathbf{infinitary}(\mathbf{-\infty}) && \text{if } x = -1\\
&= no\_result_F(x) && \text{otherwise}
\end{aligned}$$

NOTE – **underflow** is explicitly avoided.

5.3.11.10 Inverse hyperbolic cotangent

The $arccoth^*_F$ approximation helper function:

$arccoth^*_F : F \to \mathcal{R}$

$arccoth^*_F(x)$ returns a close approximation to $\mathrm{arccoth}(x)$ in $\mathcal{R}$, with maximum error $max\_error\_tanh_F$.

A further requirement on the $arccoth^*_F$ approximation helper function is:

$arccoth^*_F(-x) = -arccoth^*_F(x)$ if $x \in F$

The $arccoth_F$ operation:

$arccoth_F : F \to F \cup \{\mathbf{underflow}, \mathbf{infinitary}, \mathbf{invalid}\}$

$$
\begin{array}{lll}
arccoth_F(x) & = result^*_F(arccoth^*_F(x), nearest_F) & \\
 & & \text{if } x \in F \text{ and } |x| > 1 \\
 & = \mathbf{infinitary}(+\infty) & \text{if } x = 1 \\
 & = \mathbf{infinitary}(-\infty) & \text{if } x = -1 \\
 & = \mathbf{-0} & \text{if } x = -\infty \\
 & = 0 & \text{if } x = +\infty \\
 & = no\_result_F(x) & \text{otherwise}
\end{array}
$$

NOTE – There is no **underflow** for this operation for most kinds of floating point types, e.g. IEC 60559 ones.

5.3.11.11 Inverse hyperbolic secant

The $arcsech^*_F$ approximation helper function:

$arcsech^*_F : F \to \mathcal{R}$

$arcsech^*_F(x)$ returns a close approximation to arcsech(x) in $\mathcal{R}$, with maximum error $max\_error\_tanh_F$.

The $arcsech_F$ operation:

$arcsech_F : F \to F \cup \{\mathbf{infinitary}, \mathbf{invalid}\}$

$$
\begin{array}{lll}
arcsech_F(x) & = result^*_F(arcsech^*_F(x), nearest_F) & \\
 & & \text{if } x \in F \text{ and } 0 < x \leqslant 1 \\
 & = \mathbf{infinitary}(+\infty) & \text{if } x \in \{\mathbf{-0}, 0\} \\
 & = no\_result_F(x) & \text{otherwise}
\end{array}
$$

5.3.11.12 Inverse hyperbolic cosecant

The $arccsch^*_F$ approximation helper function:

$arccsch^*_F : F \to \mathcal{R}$

$arccsch^*_F(x)$ returns a close approximation to arccsch(x) in $\mathcal{R}$, with maximum error $max\_error\_tanh_F$.

A further requirement on the $arccsch^*_F$ approximation helper function is:

$arccsch^*_F(-x) = -arccsch^*_F(x)$ if $x \in F$

The $arccsch_F$ operation:

$arccsch_F : F \to F \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$$
\begin{array}{lll}
arccsch_F(x) & = result^*_F(arccsch^*_F(x), nearest_F) & \\
 & & \text{if } x \in F \text{ and } x \neq 0 \\
 & = div_F(1, x) & \text{if } x \in \{-\infty, \mathbf{-0}, 0, +\infty\} \\
 & = no\_result_F(x) & \text{otherwise}
\end{array}
$$

NOTE – There is no **underflow** for this operation for most kinds of floating point types, e.g. IEC 60559 ones.

5.4 Operations for conversion between numeric datatypes

Numeric conversion between different representation forms for integer and fractional values can take place under a number of different circumstances. E.g.:

a) explicit or implicit conversion between different numeric datatypes conforming to part 1;

b) explicit or implicit conversion between different numeric datatypes only one of which conforms to part 1;

c) explicit or implicit conversion between a character string and a numeric datatype.

The latter includes outputting a numeric value as a character string, inputting a numeric value from a character string source, and converting a numeral in the source program to a value in a numeric datatype (see clause 5.5). This part covers only the cases where at least one of the source and target is a numeric datatype conforming to part 1.

When a character string is involved as either source or target of a conversion, this part does not specify the lexical syntax for the numerals parsed or formed. A binding standard should specify the lexical syntax or syntaxes for these numerals, and, when appropriate, how the lexical syntax for the numerals can be altered. This could include which script for the digits to use in a position system (Latin-Arabic digits, Arabic-Indic digits, traditional Thai digits, etc.). With the exception of the radix used in numerals expressing fractional values, differences in lexical syntactic details that do not affect the value in $\mathcal{R}$ denoted by the numerals should not affect the result of the conversion.

Character string representations for integer values can include representations for $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, and quiet NaNs. Character string representations for floating point and fixed point values should have formats for $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, and quiet NaNs. For both integer and floating point values, character strings that are not numerals nor special values according to the lexical syntax used, shall be regarded as signalling NaNs when used as source of a numerical conversion.

For the cases where one of the datatypes involved in the conversion does not conform to part 1, the values of some numeric datatype parameters need to be inferred. For integers, one need to infer the value for *bounded*, and if that is **true** then also values for *maxint* and *minint*, and for string formats also the *radix*. For floating point values, one need to infer the values for r, p, and *emax* or *emin*. In case a precise determination is not possible, values that are 'safe' for that instance should be used. 'Safe' values for otherwise undetermined inferred parameters are such that

a) monotonicity of the conversion function is not affected,

b) the error in the conversion does not exceed that specified by the maximum error parameter (see below),

c) if the value resulting from the conversion is converted back to the source datatype by a conversion conforming to this part, the original value should be regenerated if possible, and

d) overflow and underflow are avoided if possible.

If, and only if, a specified infinite special value result cannot be represented in the target datatype, the infinity result shall be interpreted as implying the **infinitary** notification. If, and only if, a specified NaN special value result cannot be represented in the target datatype, the NaN result shall be interpreted as implying the **invalid** notification. If, and only if, a specified

$\mathbf{-0}$ special value result cannot be represented in the target datatype, the $\mathbf{-0}$ result shall be interpreted as 0.

5.4.1 Integer to integer conversions

Let I and I' be non-special value sets for integer datatypes. At least one of the datatypes corresponding to I and I' conforms to part 1.

$$convert_{I \to I'} : I \to I' \cup \{\mathbf{overflow}\}$$

$$\begin{aligned} convert_{I \to I'}(x) &= result_{I'}(x) && \text{if } x \in I \\ &= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ &= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN} \\ &= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN} \end{aligned}$$

NOTE – If both I and I' are conforming to part 1, then this conversion is covered by part 1. This operation generalises the $cvt_{I \to I'}$ of part 1, since only one of the integer datatypes in the conversion need be conforming to part 1.

5.4.2 Floating point to integer conversions

Let I be the non-special value set for an integer datatype conforming to part 1. Let F be the non-special value set for a floating point datatype conforming to part 1.

$$floor_{F \to I} : F \to I \cup \{\mathbf{overflow}\}$$

$$\begin{aligned} floor_{F \to I}(x) &= result_I(\lfloor x \rfloor) && \text{if } x \in F \\ &= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ &= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN} \\ &= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN} \end{aligned}$$

$$rounding_{F \to I} : F \to I \cup \{\mathbf{-0}, \mathbf{overflow}\}$$

$$\begin{aligned} rounding_{F \to I}(x) & \\ &= result_I(\mathrm{round}(x)) && \text{if } x \in F \text{ and } (x \geqslant 0 \text{ or } \mathrm{round}(x) \neq 0) \\ &= \mathbf{-0} && \text{if } x \in F \text{ and } x < 0 \text{ and } \mathrm{round}(x) = 0 \\ &= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ &= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN} \\ &= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN} \end{aligned}$$

$$ceiling_{F \to I} : F \to I \cup \{\mathbf{-0}, \mathbf{overflow}\}$$

$$\begin{aligned} ceiling_{F \to I}(x) &= result_I(\lceil x \rceil) && \text{if } x \in F \text{ and } (x \geqslant 0 \text{ or } \lceil x \rceil \neq 0) \\ &= \mathbf{-0} && \text{if } x \in F \text{ and } x < 0 \text{ and } \lceil x \rceil = 0 \\ &= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ &= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN} \\ &= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN} \end{aligned}$$

5.4.3 Integer to floating point conversions

Let I be the non-special value set for an integer datatype. Let F be the non-special value set for a floating point datatype. At least one of the source and target datatypes is conforming to part 1.

$$convert_{I\to F} : I \to F \cup \{\mathbf{overflow}\}$$

$$\begin{array}{lll} convert_{I\to F}(x) & = result_F(x, nearest_F) & \text{if } x \in I \\ & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = \mathbf{qNaN} & \text{if } x \text{ is a quiet NaN} \\ & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN} \end{array}$$

NOTE – When both I and F conform to part 1, integer to nearest floating point conversions are covered by part 1. In this case the operations $cvt_{I\to F}$ and $convert_{I\to F}$ are identical.

5.4.4 Floating point to floating point conversions

Define the least radix function, lb, defined for arguments that are greater than 0:

$$lb : \mathcal{Z} \to \mathcal{Z}$$

$$lb(r) = \min\{n \in \mathcal{Z} \mid n \geqslant 1 \text{ and there is an } m \in \mathcal{Z} \text{ such that } r = n^m\}$$

Let F, F', and F'' be non-special value sets for floating point datatypes. At least one of the source and target datatypes in the conversion conforms to part 1.

There shall be a $max\_error\_convert_{F'}$ parameter that gives the maximum error when converting from F to F' and $lb(r_F) \neq lb(r_{F'})$. The $max\_error\_convert_{F'}$ parameter shall have the value 0.5. If the binding standard requires that this parameter has the value 0.5 (see annex A), this parameter need not be made available for programs.

If $lb(r_F) = lb(r_{F'})$, the maximum error shall be 0.5 ulp when converting from F to F', even when the implementation is only partially conforming (see Annex A), but this is not reflected in any parameter.

The $convert^*_{F\to F'}$ approximation helper functions:

$$convert^*_{F\to F'} : F \to \mathcal{R}$$

$convert^*_{F\to F'}(x)$ returns a close approximation to x in $\mathcal{R}$, with maximum error $max\_error\_convert_{F'}$.

NOTE 1 – With the maximum error 0.5 ulp, this and the below conversion helper functions are not really needed. However, Annex A allows for partial conformity, such that the maximum error for these helper functions may be greater than 0.5 ulp.

Further requirements on the $convert^*_{F\to F'}$ approximation helper functions are:

$$\begin{array}{ll} convert^*_{F\to F'}(x) = x & \text{if } x \in \mathcal{Z} \cap F \\ convert^*_{F\to F'}(x) > 0 & \text{if } x \in F \text{ and } x > 0 \\ convert^*_{F\to F'}(-x) = -convert^*_{F\to F'}(x) & \text{if } x \in F \\ convert^*_{F\to F'}(x) \leqslant convert^*_{F\to F'}(y) & \text{if } x, y \in F \text{ and } x < y \end{array}$$

Relationship to other floating point to floating point conversion approximation helper functions for conversion operations in the same library shall be:

$$convert^*_{F\to F'}(x) = convert^*_{F''\to F'}(x) \qquad \text{if } lb(r_{F''}) = lb(r_F) \text{ and } x \in F \cap F''$$

The $convert_{F\to F'}$ operation:

$$convert_{F\to F'} : F \to F' \cup \{\mathbf{overflow}, \mathbf{underflow}\}$$

$$\begin{aligned}
\mathit{convert}_{F\to F'}(x) &= \mathit{result}_{F'}(x, \mathit{nearest}_{F'}) && \text{if } x \in F \text{ and } lb(r_F) = lb(r_{F'})\\
&= \mathit{result}^*_{F'}(\mathit{convert}^*_{F\to F'}(x), \mathit{nearest}_{F'}) && \\
& && \text{if } x \in F \text{ and } lb(r_F) \neq lb(r_{F'})\\
&= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\}\\
&= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN}\\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN}
\end{aligned}$$

NOTES

2 Modern techniques allow, on the average, efficient conversion with a maximum error of 0.5 ulp even when the radices differ. C99 [17], for instance, requires that all floating point value conversion is done with a maximum error of 0.5 ulp.

3 IEC 60559 requirements imply that the $\mathit{max\_error\_convert}_{F'}$ parameter has a value $\leqslant 0.97$. Such a large maximum error for the conversion is only partially conforming. See Annex A.

4 When the maximum error is 0.5, the conversion helper function above can be the identity function.

5 When both datatypes conform to part 1, and the radices for both of these floating point datatypes are the same, floating point to nearest floating point conversions are covered by part 1. In this case the operations $\mathit{cvt}_{F\to F'}$ and $\mathit{convert}_{F\to F'}$ are identical.

5.4.5 Floating point to fixed point conversions

Let F be the non-special value set for a floating point datatype conforming to part 1. Let D be the non-special value set for a fixed point datatype.

A fixed point datatype D is a subset of $\mathcal{R}$, characterised by a radix, $r_D \in \mathcal{Z}$ ($\geqslant 2$), a density, $d_D \in \mathcal{Z}$ ($\geqslant 0$), and if it is bounded, a maximum positive value, $\mathit{dmax}_D \in D^*$ ($\geqslant 1$). Given these values, the following sets are defined:

$$D^* = \{n/(r_D^{d_D}) \mid n \in Z\}$$

$$\begin{aligned}
D &= D^* && \text{if } D \text{ is not bounded}\\
D &= D^* \cap [-\mathit{dmax}_D, \mathit{dmax}_D] && \text{if } D \text{ is bounded}
\end{aligned}$$

NOTE 1 – D corresponds to **scaled(**r_D**,** d_D**)** in ISO/IEC 11404 *Language independent datatypes (LID)* [10]. LID has no parameter corresponding to dmax_D even when the datatype is bounded.

The fixed point rounding helper function:

$$\mathit{nearest}_D : \mathcal{R} \to D^*$$

is the rounding function that rounds to nearest, ties round to even last digit.

The fixed point result helper function, result_D, is like result_F, but for a fixed point datatype. It will return **overflow** if the rounded result is not representable:

$$\mathit{result}_D : \mathcal{R} \times (\mathcal{R} \to D^*) \to D \cup \{\mathbf{-0}, \mathbf{overflow}\}$$

$$\begin{aligned}
\mathit{result}_D(x, \mathit{rnd}) &= \mathit{rnd}(x) && \text{if } \mathit{rnd}(x) \in D \text{ and } (\mathit{rnd}(x) \neq 0 \text{ or } x \geqslant 0)\\
&= \mathbf{-0} && \text{if } \mathit{rnd}(x) = 0 \text{ and } x < 0\\
&= \mathbf{overflow} && \text{if } x \in \mathcal{R} \text{ and } \mathit{rnd}(x) \notin D
\end{aligned}$$

There shall be a $\mathit{max\_error\_convert}_D$ parameter that gives the maximum error when converting from F to D and $lb(r_F) \neq lb(r_D)$. The $\mathit{max\_error\_convert}_D$ parameter shall have the value 0.5. If the binding standard requires that this parameter has the value 0.5 (see annex A), this parameter need not be made available for programs.

If $lb(r_F) = lb(r_D)$, the maximum error shall be 0.5 ulp when converting from F to D, even when the implementation is only partially conforming (see Annex A), but this is not reflected in any parameter.

The $convert^*_{F \to D}$ approximation helper function:

$$convert^*_{F \to D} : F \to \mathcal{R}$$

$convert^*_{F \to D}(x)$ returns a close approximation to x in $\mathcal{R}$, with maximum error $max\_error\_convert_D$.

Further requirements on the $convert^*_{F \to D}$ approximation helper functions are:

$$\begin{array}{ll}
convert^*_{F \to D}(x) = x & \text{if } x \in \mathcal{Z} \cap F \\
convert^*_{F \to D}(x) > 0 & \text{if } x \in F \text{ and } x > 0 \\
convert^*_{F \to D}(-x) = -convert^*_{F \to D}(x) & \text{if } x \in F \\
convert^*_{F \to D}(x) \leqslant convert^*_{F \to D}(y) & \text{if } x, y \in F \text{ and } x < y
\end{array}$$

Relationship to other floating point to fixed point conversion approximation helper functions for conversion operations in the same library shall be:

$$convert^*_{F \to D}(x) = convert^*_{F'' \to D}(x) \qquad \text{if } lb(r_{F''}) = lb(r_F) \text{ and } x \in F \cap F''$$

The $convert_{F \to D}$ operation:

$$convert_{F \to D} : F \to D \cup \{\mathbf{-0}, \mathbf{overflow}\}$$

$$\begin{array}{lll}
convert_{F \to D}(x) & = result_D(x, nearest_D) & \text{if } x \in F \text{ and } lb(r_F) = lb(r_D) \\
& = result_D(convert^*_{F \to D}(x), nearest_D) & \\
& & \text{if } x \in F \text{ and } lb(r_F) \neq lb(r_D) \\
& = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
& = \mathbf{qNaN} & \text{if } x \text{ is a quiet NaN} \\
& = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN}
\end{array}$$

NOTES

2 The datatype D need not be visible in the programming language. D may be a subtype of strings, according to some format. Even so, no datatype for strings need be present in the programming language.

3 This covers, among other things, "output" of floating point datatype values, to fixed point string formats. E.g. a binding may say that `float_to_fixed_string(`x`, `m`, `n`)` is bound to $convert_{F \to S_{m,n}}(x)$ where $S_{m,n}$ is strings of length m, representing fixed point values in radix 10 with n decimals. The binding should also detail how NaNs, signed zeroes and infinities are represented in $S_{m,n}$, as well as the precise format of the strings representing ordinary values. (Note that if the length of the target string is limited, the conversion may overflow.)

4 IEC 60559 requirements imply that the $max\_error\_convert_D$ parameter has a value $\leqslant 0.97$. Such a large maximum error for the conversion is only partially conforming. See Annex A.

5 When the maximum error is 0.5, the conversion helper function above can be the identity function.

5.4.6 Fixed point to floating point conversions

Let F be the non-special value set for a floating point datatype conforming to part 1. Let D and D' be the non-special value set for fixed point datatypes.

The $convert^*_{D \to F}$ approximation helper function:

$$convert^*_{D \to F} : D \to \mathcal{R}$$

$convert^*_{D\to F}(x)$ returns a close approximation to x in $\mathcal{R}$, with maximum error $max\_error\_convert_F$.

Further requirements on the $convert^*_{D\to F}$ approximation helper functions are:

$$\begin{array}{ll}
convert^*_{D\to F}(x) = x & \text{if } x \in \mathcal{Z} \cap D \\
convert^*_{D\to F}(x) > 0 & \text{if } x \in D \text{ and } x > 0 \\
convert^*_{D\to F}(-x) = -convert^*_{D\to F}(x) & \text{if } x \in D \\
convert^*_{D\to F}(x) \leqslant convert^*_{D\to F}(y) & \text{if } x, y \in D \text{ and } x < y
\end{array}$$

Relationship to other floating point and fixed point to floating point conversion approximation helper functions for conversion operations in the same library shall be:

$$\begin{array}{ll}
convert^*_{D\to F}(x) = convert^*_{D'\to F}(x) & \text{if } lb(r_{D'}) = lb(r_D) \text{ and } x \in D \cap D' \\
convert^*_{D\to F}(x) = convert^*_{F'\to F}(x) & \text{if } lb(r_{F'}) = lb(r_D) \text{ and } x \in D \cap F'
\end{array}$$

The $convert_{D\to F}$ operation:

$convert_{D\to F} : D \to F \cup \{\mathbf{overflow}, \mathbf{underflow}\}$

$$\begin{array}{lll}
convert_{D\to F}(x) & = result_F(x, nearest_F) & \text{if } x \in D \text{ and } lb(r_D) = lb(r_F) \\
 & = result^*_F(convert^*_{D\to F}(x), nearest_F) & \\
 & & \text{if } x \in D \text{ and } lb(r_D) \neq lb(r_F) \\
 & = x & \text{if } x \in \{-\infty, -\mathbf{0}, +\infty\} \\
 & = \mathbf{qNaN} & \text{if } x \text{ is a quiet NaN} \\
 & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN}
\end{array}$$

NOTES

1 This covers, among other things, “input” of floating point datatype values, from fixed point string formats. E.g. a binding may say that `string_to_float(`s`)` is bound to $convert_{S_{m,n}\to F}(s)$ where $S_{m,n}$ is strings of length m, where m is the length of s, and n is the number of digits after the “decimal symbol” represented in $S_{m,n}$, as well as the precise format of the strings representing ordinary values.

2 When the maximum error is 0.5, the conversion helper function above can be the identity function.

5.5 Numerals as operations in a programming language

NOTE – Numerals in strings, or input, is covered by the conversion operations in clause 5.4.

Each numeral is a parameterless operation. Thus, this clause introduces a very large number of operations, since the number of numerals is in principle infinite.

5.5.1 Numerals for integer datatypes

Let I' be a non-special value set for integer numerals for the datatype corresponding to I.

An integer numeral, denoting an abstract value n in $I' \cup \{-\mathbf{0}, +\infty, -\infty, \mathbf{qNaN}, \mathbf{sNaN}\}$, for an integer datatype with non-special value set I, shall result in

$convert_{I'\to I}(n)$

For each integer datatype conforming to part 1 and made directly available, there shall be integer numerals with radix 10.

For each radix for numerals made available for a bounded integer datatype with non-special value set I, there shall be integer numerals for all non-negative values of I. For each radix for numerals made available for an unbounded integer datatype, there shall be integer numerals for all non-negative integer values smaller than 10^{20}.

For each integer datatype made directly available and that may have special values:

a) There should be a numeral for positive infinity. There shall be a numeral for positive infinity if there is a positive infinity in the integer datatype.

b) There should be numerals for quiet and signalling NaNs.

5.5.2 Numerals for floating point datatypes

Let D be a non-special value set for fixed point numerals for the datatype corresponding to F. Let F' be a non-special value set for floating point numerals for the datatype corresponding to F.

A fixed point numeral, denoting an abstract value x in $D \cup \{\mathbf{-0}, \mathbf{+\infty}, \mathbf{-\infty}, \mathbf{qNaN}, \mathbf{sNaN}\}$, for a floating point datatype with non-special value set F, shall result in

$convert_{D \to F}(x)$

A floating point numeral, denoting an abstract value x in $F' \cup \{\mathbf{-0}, \mathbf{+\infty}, \mathbf{-\infty}, \mathbf{qNaN}, \mathbf{sNaN}\}$, for a floating point datatype with non-special value set F, shall result in

$convert_{F' \to F}(x)$

For each floating point datatype conforming to part 1 and made directly available, there should be radix 10 floating point numerals, and there shall be radix 10 fixed point numerals.

For each radix for fixed point numerals made available for a floating point datatype, there shall be numerals for all bounded precision and bounded range expressible non-negative values of $\mathcal{R}$. At least a precision (d_D) of 20 should be available. At least a range ($dmax_D$) of 10^{20} should be available.

For each radix for floating point numerals made available for a floating point datatype with non-special value set F, there shall be numerals for all bounded precision and bounded range expressible non-negative values of $\mathcal{R}$. The precision and range bounds for the numerals shall be large enough to allow all non-negative values of F to be reachable.

For each floating point datatype made directly available and that may have special values:

a) There should be a numeral for positive infinity. There shall be a numeral for positive infinity if there is a positive infinity in the floating point datatype.

b) There should be numerals for quiet and signalling NaNs.

The conversion operations used for numerals as operations should be the same as those used by default for converting strings to values in conforming integer or floating point datatypes.

6 Notification

Notification is the process by which a user or program is informed that an arithmetic operation cannot return a suitable numeric result. Specifically, a notification shall occur when any arithmetic operation returns an exceptional value. Notification shall be performed according to the requirements of clause 6 of part 1.

An implementation shall not give notifications for operations conforming to this part, unless the specification requires that an exceptional value results for the given arguments.

The default method of notification should be recording of indicators.

6.1 Continuation values

If notifications are handled by a recording of indicators, in the event of notification the implementation shall provide a *continuation value* to be used in subsequent arithmetic operations. Continuation values may be in I or F (as appropriate), or be special values ($\mathbf{-0}$, $\mathbf{-\infty}$, $\mathbf{+\infty}$, or a **qNaN**).

Floating point datatypes that satisfy the requirements of IEC 60559 have special values in addition to the values in F. These are: $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, *signalling* **NaN**s (**sNaN**), and *quiet* **NaN**s (**qNaN**). Such values may be passed as arguments to operations, and used as results or continuation values. Floating point types that do not fully conform to IEC 60559 can also have values corresponding to $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, or **NaN**.

Continuation values of $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, and **NaN** are required only if the parameter iec_559_F has the value **true**. If the implementation can represent such special values in the result datatype, they should be used according to the specifications in this part.

7 Relationship with language standards

A computing system often provides some of the operations specified in this part within the context of a programming language. The requirements of the present standard shall be in addition to those imposed by the relevant programming language standards.

This part does not define the syntax of arithmetic expressions. However, programmers need to know how to reliably access the operations specified in this part.

NOTE 1 – Providing the information required in this clause is properly the responsibility of programming language standards. An individual implementation would only need to provide details if it could not cite an appropriate clause of the language or binding standard.

An implementation shall document the notation that should be used to invoke an operation specified in this part and made available. An implementation should document the notation that should be used to invoke an operation specified in this part and that could be made available.

NOTE 2 – For example, the radian arc sine operation for an argument x ($arcsin_F(x)$) might be invoked as

`arcsin(x)`	in Pascal [27] and Ada [11]
`asin(x)`	in C [17] and Fortran [22]
`(asin x)`	in Common Lisp [42] and ISLisp [24]
`function asin(x)`	in COBOL [19]

with a suitable expression for the argument (x).

An implementation shall document the semantics of arithmetic expressions in terms of compositions of the operations specified in clause 5 of this part and in clause 5 of part 1.

NOTE 3 – An arithmetic expression might not be executed as written.

For example, if x is declared to be single precision (`SP`) floating point, and calculation is done in single precision, then the expression

```
arcsin(x)
```

might translate to

$arcsin_{SP}(\texttt{x})$

If the language in question did all computations in double precision (`DP`) floating point, the above expression might translate to

$arcsin_{DP}(convert_{SP \to DP}(\texttt{x}))$

Alternatively, if `x` was declared to be an integer, and the expected result datatype is single precision float, the above expression might translate to

$convert_{DP \to SP}(arcsin_{DP}(convert_{I \to DP}(\texttt{x})))$

Compilers often "optimize" code as part of compilation. Thus, an arithmetic expression might not be executed as written. An implementation shall document the possible transformations of arithmetic expressions (or groups of expressions) that it permits. Typical transformations include

a) Insertion of operations, such as datatype conversions or changes in precision.

b) Replacing operations (or entire subexpressions) with others, such as "`cos(-x)`" → "`cos(x)`" (exactly the same result) or "`pi - arccos(x)`" → "`arccos(-x)`" (more accurate result) or "`exp(x)-1`" → "`expm1(x)`" (more accurate result if $x > -1$, less accurate result if $x < -1$, different notification behaviour).

c) Evaluating constant subexpressions.

d) Eliminating unneeded subexpressions.

Only transformations which alter the semantics of an expression (the values produced, and the notifications generated) need be documented. Only the range of permitted transformations need be documented. It is not necessary to describe the specific choice of transformations that will be applied to a particular expression.

The textual scope of such transformations shall be documented, and any mechanisms that provide programmer control over this process should be documented as well.

NOTE 4 – It is highly desirable that programming languages intended for numerical use provide means for limiting the transformations applied to particular arithmetic expressions.

8 Documentation requirements

In order to conform to this part, an implementation shall include documentation providing the following information to programmers.

NOTE – Much of the documentation required in this clause is properly the responsibility of programming language or binding standards. An individual implementation would only need to provide details if it could not cite an appropriate clause of the language or binding standard.

a) A list of the provided operations that conform to this part.

b) For each maximum error parameter, the value of that parameter or definition of that parameter function. Only maximum error parameters that are relevant to the provided operations need be given.

c) The value of the parameters $big\_angle\_r_F$ and $big\_angle\_u_F$. Only big angle parameters that are relevant to the provided operations need be given.

d) For the $nearest_F$ function, the rule used for rounding halfway cases, unless iec_559_F is fixed to **true**.

e) For each conforming operation, the continuation value provided for each notification condition. Specific continuation values that are required by this part need not be documented. If the notification mechanism does not make use of continuation values (see clause 6), continuation values need not be documented.

f) For each conforming operation, how the results depend on the rounding mode, if rounding modes are provided. Operations may be insensitive to the rounding mode, or sensitive to it, but even then need not heed the rounding mode.

g) For each conforming operation, the notation to be used for invoking that operation.

h) For each maximum error parameter, the notation to be used to access that parameter.

i) The notation to be used to access the parameters $big\_angle\_r_F$ and $big\_angle\_u_F$.

j) For each of the provided operations where this part specifies a relation to another operation specified in this part, the binding for that other operation.

k) For numerals conforming to this part, which available string conversion operations, including reading from input, give exactly the same conversion results, even if the string syntaxes for 'internal' and 'external' numerals are different.

Since the integer and floating point datatypes used in conforming operations shall satisfy the requirements of part 1, the following information shall also be provided by any conforming implementation.

l) The means for selecting the modes of operation that ensure conformity.

m) The translation of arithmetic expressions into combinations of the operations provided by any part of ISO/IEC 10967, including any use made of higher precision. (See clause 7 of part 1.)

n) The methods used for notification, and the information made available about the notification. (See clause 6 of part 1.)

o) The means for selecting among the notification methods, and the notification method used in the absence of a user selection. (See clause 6.3 of part 1.)

p) When "recording of indicators" is the method of notification, the datatype used to represent *Ind* (see clause 6.1.2 of part 1), the method for denoting the values of *Ind*, and the notation for invoking each of the "indicator" operations. E is the set of notification indicators. The association of values in *Ind* with subsets of E must be clear. In interpreting clause 6.1.2 of part 1, the set of indicators E shall be interpreted as including all exceptional values listed in the signatures of conforming operations. In particular, E may need to contain **infinitary** and **absolute_precision_underflow**.

Annex A
(normative)

Partial conformity

If an implementation of an operation fulfills all relevant requirements according to the main normative text in this part, except the ones relaxed in this Annex, the implementation of that operation is said to *partially conform* to this part.

Conformity to this part shall not be claimed for operations that only fulfill partial conformity.

Partial conformity shall not be claimed for operations that relax other requirements than those relaxed in this Annex.

A.1 Maximum error relaxation

This part has the following maximum error requirements for conformity.

$$max\_error\_hypot_F \in [0.5, 1]$$

$$max\_error\_exp_F \in [0.5, 1.5 \cdot rnd\_error_F]$$
$$max\_error\_power_F \in [0.5, 2 \cdot rnd\_error_F]$$

$$max\_error\_rad_F = 0.5$$
$$max\_error\_sin_F \in [0.5, 1.5 \cdot rnd\_error_F]$$
$$max\_error\_tan_F \in [0.5, 2 \cdot rnd\_error_F]$$

$$max\_error\_sinu_F : F \to F \cup \{\mathbf{invalid}\}$$
$$max\_error\_tanu_F : F \to F \cup \{\mathbf{invalid}\}$$

$$max\_error\_sinh_F \in [0.5, 2 \cdot rnd\_error_F]$$
$$max\_error\_tanh_F \in [0.5, 2 \cdot rnd\_error_F]$$

$$max\_error\_convert_F = 0.5$$
$$max\_error\_convert_D = 0.5$$

For $u \in G_F$, the $max\_error\_sinu_F(u)$ parameter shall have a value in the interval $[0.5, 2 \cdot max\_error\_sin_F]$, and the $max\_error\_tanu_F(u)$ parameter shall have a value in the interval $[0.5, 2 \cdot max\_error\_tan_F]$. For $u \in T$, the $max\_error\_sinu_F(u)$ parameter shall be equal to $max\_error\_sin_F$, and the $max\_error\_tanu_F(u)$ parameter shall be equal to $max\_error\_tan_F$, for the same library.

In a partially conforming implementation the maximum error parameters may be greater than what is specified by this part. The maximum error parameter values given by an implementation shall still adequately reflect the accuracy of the relevant operations, if a claim of partial conformity is made.

A partially conforming implementation shall document which maximum error parameters have greater values than specified by this part, and their values.

A.2 Extra accuracy requirements relaxation

This part has a number of extra accuracy requirements. These are detailed in the paragraphs beginning "Further requirements on the op^*_F approximation helper function are:".

In a partially conforming implementation these further requirements need not be fulfilled. The values returned must still be within the maximum error bounds that are given by the maximum error parameters, if a claim of partial conformity is made.

The extra accuracy requirements together with the sign and monotonicity requirements imply a number of requirements that are not stated explicitly, due to that they are implied. Removing one or more of thee given requirements may thus remove some weaker requirements that were not intended to be removed. Some of the remaining weaker requirements may need to be stated explicitly if a stronger requirement is removed.

A partially conforming implementation shall document which extra accuracy requirements are not fulfilled by the implementation, and which weaker requirements that are still fulfilled.

A.3 Relationships to other operations relaxation

This part has a number of requirements giving relations to other operations. These are detailed in the paragraphs beginning with wordings like "Relationship to the op^*_F approximation helper function for operations in the same library shall be:".

In a partially conforming implementation these relationships need not be fulfilled. The values returned must still be within the maximum error bounds that are given by the values provided for the maximum error parameters, if a claim of partial conformity is made.

A partially conforming implementation shall document which operation relationships are not fulfilled by the implementation.

A.4 Very-close-to-axis angular normalisation relaxation

This part requires, explicitly and by implication, that angular normalisation (sometimes called argument reduction) is done so that the (intermediate or explicit) result is accurate within less than an ulp. For angular values, especially in radians, that denote an angle very close to an axis, that requires extra high precision in the calculation of the normalised value.

In a partially conforming implementation the accuracy requirements for angular normalisation for angles that are very close to an axis need not be fulfilled.

A partially conforming implementation shall document which trigonometric operations and for which (small) intervals near axes angular values, that are not so large that **absolute_precision_underflow** notifications would be the result, the angular normalisation accuracy requirements are not fulfilled by the implementation. The implementation shall also document how large the absolute error for angular normalisation is also for angles that are in those intervals very near an axis. It may be appropriate for a binding to specify one or more parameters describing this relaxation if this relaxation is allowed by a binding. The maximum error parameter values given by an implementation shall still adequately reflect the accuracy of the relevant trigonometric operations for angular values outside of those very-near-axis intervals, if a claim of partial conformity is made.

A.5 Part 1 requirements relaxation

Part 2 depends on the datatypes and operations specified in part 1. Part 1 allows for partial conformity. Part 2 operations may thus be only partially conforming if a relevant datatype or part 1 operation is only partially conforming to part 1.

Annex B
(informative)

Rationale

This annex explains and clarifies some of the ideas behind *Information technology – Language independent arithmetic – Part 2: Elementary numerical functions* (LIA-2). This allows the standard itself to be more concise. The clause numbering matches that of the standard, although additional clauses have been added.

B.1 Scope

The scope of LIA-2 includes the traditional arithmetic operations, that are not already covered by LIA-1, usually provided in programming languages. This includes operations that are numeric approximations to real elementary functions. Even though these operations usually are implemented in software rather than hardware they are still to be regarded as *atomic* in the sense that they are never (as seen by the user) interrupted by an intermediate notifiacation.

B.1.1 Inclusions

LIA-2 is intended to define the meaning of some operations on integer and floating point types as specified in LIA-1 (ISO/IEC 10967-1), in addition to the operations specified in LIA-1. LIA-2 does not specify operations for any additional arithmetic datatypes, though fixed point datatypes are used in some of the specifications for conversion operations.

The specifications for the operations covered by LIA-2 are given in sufficient detail to

a) support detailed and accurate numerical analysis of arithmetic algorithms,

b) enable a precise determination of conformity or non-conformity, and

c) prevent exceptions (like overflow) from going undetected.

LIA-2 only covers operations that involve integer or floating point datatypes, as specified in LIA-1, and in some cases also a Boolean datatype, but then only as result. In order to include also fixed point string formats for floating point values, fixed point datatypes are also involved in some of the LIA-2 conversion operations.

The operations covered by LIA-2 are often to some extent covered by programming language standards, like the operations `sin`, `cos`, `tan`, `arctan`, and so on. Annex C also surveys which operations are already covered by various programming languages.

LIA-2 includes some operations that are not (yet) common in programming languages. Like operations to normalise angular values, and to convert angular values between different angular units. These operations are closely related to the other operations included in LIA-2, and these operations are non-trivial to implement with high accuracy. The angular normalisation operations are useful to keep high accuracy in the angular values used when increasing angular values are used.

LIA-2 does in no way prevent language standards or implementations including further arithmetic operations, other variations of included arithmetic operations, or the inclusion of further arithmetic datatypes, like rational number or fixed point datatypes. Some of these may become the topic of standardization in other parts of LIA.

B.1.2 Exclusions

LIA-2 is not concerned with techniques for the *implementation* of numerical functions. Even when an LIA-2 operation specification is made in terms of other LIA-1 or LIA-2 operations, that does *not* imply a requirement that an implementation implements the operation in terms of those other operations. It is sufficient that the result (returned value or returned continuation value, and exception behaviour) is *as if* it was implemented in terms of those other operations.

LIA-2 does not provide specifications for operations which involve no arithmetic processing, like assignment and parameter passing, though any implicit conversions done in association with such operations are in scope. The implicit conversions should be made available to the programmer as explicit conversions.

LIA-2 does not cover operations for the support of domains such as linear algebra, statistics, and symbolic processing. Such domains deserve *separate* standardization, if standardized.

LIA-2 does not cover how to represent numeric values, internally (as bit patterns) or externally (as character strings).

B.2 Conformity

Conformity to this standard is dependent on the existence of language binding standards. Each programming language committee (or other organization responsible for a programming language or other specification to which LIA-1 and LIA-2 may apply) is encouraged to produce a binding covering at least those operations already required by the programming language (or similar) and also specified in LIA-2.

The term "programming language" is here used in a generalised sense to include other computing entities such as calculators, spread sheets, page description languages, web-script languages, and database query languages to the extent that they provide the operations covered by LIA-2.

A conforming system consists of an implementation (which obeys the requirements) together with documentation which shows how the implementation conforms to the standard. This documentation is vital since it gives crucial characteristics of the system, such as the range for trigonometric operations, and the accuracy of the operations.

The binding of LIA-2 facilities to a particular programming language should be as natural as possible. Existing language syntax and features should be used for operations, parameters, notification, and so on. For example, if a language expresses application of cosine as "`cos(x)`," then LIA-2 cosine operations cos_F should be bound to (overloaded) "`cos`" functions.

Suggestions for bindings are provided in annex C. Annex C has partial binding examples for a number of existing programming languages and LIA-2. In addition to the bindings for the operations in LIA-2, it is also necessary to provide bindings for the maximum error parameters and big angle parameters specified by LIA-2. Annex C contains suggestions for these bindings. To conform to this standard, in the absence of a binding standard, an implementation should create a binding, following the suggestions in annex C.

LIA-2 has fairly strict accuracy requirements. Annex A deals with the case that an implementation (or binding standard) conforms to most aspects of LIA-2, but not necessarily all of the accuracy requirements.

Some implementations, or binding standards, may wish to conform to most of the requirements in LIA-2, but make exceptions from the specifications given by LIA-2 in certain cases. Some of the bindings examples in annex C also exemplify, in different ways, such changes of specification. Real bindings are expected to elaborate such differences much more than in the examples given in annex C.

B.2.1 Validation

LIA-2 gives a very precise description of the operations included. This will expedite the construction of conformity tests. It is important that objective tests are available.

LIA-2 does not define any process for validating conformity.

Independent assurance of conformity to LIA-2 could be by spot checks on products by a validation suite. Alternatively, checking could be regarded as the responsibility of the vendor, who would then document the evidence supporting any claim to conformity.

B.3 Normative references

The referenced IEC 60559 standard is identical to the IEEE 754 standard and the former IEC 559 standard.

B.4 Symbols and definitions

LIA-2 uses the same specification mechanisms as LIA-1. LIA-2, however, uses helper functions to a much higher degree, in particular for the specification of the operations approximating elementary transcendental functions.

As in LIA-1, operations specified in LIA-2 are done so by cases, and in some of the cases helper functions are used. In contrast to LIA-1, LIA-2 also cover cases that involve "special values" for the floating point operations. The specification of how to handle these "special values" as arguments and results for the included operations is one of the major added-values of LIA-2.

The cases in each operation specification are non-overlapping, though there is an "otherwise" case at the end of many lists of cases.

B.4.1 Symbols

B.4.1.1 Sets and intervals

The interval notation is in common use. It has been chosen over the other commonly used interval notation (with brackets and round parentheses mixed) because the chosen notation has no risk of confusion with the pair notation.

B.4.1.2 Operators and relations

Note that all operators, relations, and other mathematical notation used in LIA-2 is used in their conventional exact mathematical sense. They are not used to stand for operations specified by IEC 60559, LIA-1, LIA-2, or, with the exception of program excerpts which are clearly marked, any programming language. For example, x/u stands for the mathematically exact result of dividing x by u, independently of whether that value is representable in any floating point datatype or not, and $x/u \neq div_F(x,u)$ is often the case. Likewise, $=$ is the mathematical equality, not the eq_F operation: $0 \neq \mathbf{-0}$, while $eq_F(0, \mathbf{-0}) = \mathbf{true}$.

B.4.1.3 Mathematical functions

The elementary functions named sin, cos, etc., used in LIA-2 are the exact mathematical functions, not any approximation. The approximations to these mathematical functions are introduced in clauses 5.3 and 5.4 and are written in a way clearly distinct from the mathematical functions. E.g., sin^*_F, cos^*_F, etc., which are unspecified (or, more precisely, partially specified) mathematical functions approximating the targeted exact mathematical functions to a specified degree; sin_F, cos_F, etc., which are the operations specified by LIA-2 based on the respective approximating function; `sin`, `cos`, etc., which are programming language names that may be bound to LIA-2 operations. `sin` and `cos` are thus very different from sin and cos.

B.4.1.4 Exceptional values

LIA-2 uses a modified set of exceptional values compared to LIA-1. Instead of LIA-1's **undefined**, LIA-2 uses **invalid** and **infinitary**. IEC 60559 distinguishes between **invalid** and **divide_by_zero** (the latter is called **infinitary** by LIA-2). The distinction is valid and should be recognised, since **infinitary** indicates that an infinite but *exact* result is (or can be, if it were available) returned, while **invalid** indicates that a result in the target datatype (extended with infinities) cannot, or should not, be returned with adequate accuracy.

LIA-1 distinguished between **integer_overflow** and **floating_overflow**. This distinction is moot, since no distinction was made between **integer_undefined** and **floating_undefined**. In addition, continuing this distinction would force LIA to start distinguishing not only **integer_overflow** and **floating_overflow**, but also **fixed_overflow**, **complex_floating_overflow**, **complex_integer_overflow**, etc. Further, there is no general consensus that maintaining this distinction is useful, and many programming languages do not require a distinction. A binding standard can still maintain distinctions of this kind, if desired.

infinitary is used for integer operations, when the operation rightfully should return an infinitary value, but no infinitary value occurs among the arguments. **infinitary** is also used for floating point operations for the same circumstances. That includes when the approximated real-valued function has a pole at the argument point.

LIA allows for three methods for handing notifications: recording of indicators, change of control flow (returnable or not), and termination of program. The LIA-2 preferred method is recording of indicators. This allows the computation to continue using the continuation values. For **underflow** and **infinitary** notifications this course of action is strongly preferred, provided that a suitable continuation value can be represented in the result datatype.

Not all occurrences of the same exceptional value need be handled the same. There may be explicit mode changes in how notifications are handled, and there may be implicit changes. For example, **invalid** without a specified continuation value may cause change of control flow (like an Ada [11] exception), while **invalid** with a specified continuation value may use recording of indicators. This should be specified by bindings or by implementations.

The operations may return any of the exceptional values **overflow**, **underflow**, **invalid**, **infinitary**, or **absolute_precision_underflow**. This does *not* imply that the implemented operations are to actually *return* any of these values. When these values are returned according to the LIA specification, that means that the implementation is to perform a notification handling for that exceptional value. If the notification handling is by recording of indicators, then what is actually returned by the implemented operation is the continuation value.

Most bindings are expected to be such that **underflow** and **infinitary** are "quietly" handled. If infinities are guaranteed to be representable, **infinitary** may even be disregarded completely, quietly returning the infinitary result without even any setting of any indicator.

B.4.1.5 Datatypes

The sequence types $[I]$ and $[F]$ appear as input datatypes to a few operations: max_seq_I, min_seq_I, gcd_seq_I, lcm_seq_I, max_seq_F, min_seq_F, $mmax\_seq_F$, and $mmin\_seq_F$.

In effect, a sequence is a finite linearly ordered collection of elements which can be indexed from 1 to the length of the sequence. Equality of two or more elements with different indices is possible. Sequences are used in LIA-2 as an abstraction of arrays, lists, other kinds of one-dimensional sequenced collections, and even variable length argument lists. As used in LIA-2 the order of the elements and number of occurrences of each element, as long as it is more than one, does not matter, so sets, multi-sets (bags), and tuples also qualify.

B.4.2 Definitions of terms

Note the LIA distinction between exceptional values, exceptions, and exception handling (handling of notification by non-returnable change of control flow; as in, e.g., Ada). LIA exceptional values are *not* the same as Ada `exceptions`, nor are they the same as IEC 60559 special values.

Note also that LIA-1 used the term denormal for what IEC 60559 and LIA-2 refer to as subnormal.

B.5 Specifications for the numerical functions

The abstract values used in the specifications are independent of datatype, just like the mathematical numbers are. That they are represented differently in, say, single precision and in double precision is out of scope for LIA-2.

The specifications in LIA-2 for floating point operations give details about certain special values (they are 'special' in that they are not in $\mathcal{R}$). These special values are commonly representable in floating point datatypes, in particular all floating point datatypes conforming to IEC 60559.

B.5.1 Basic integer operations

Integer datatypes can have infinity values as well as NaN values, and also may have a $\mathbf{-0}$. A corresponding I must, however, be a subset of $\mathcal{Z}$. $\mathbf{-0}$ is commonly available when the integer datatype is represented using radix-minus-1-complement, e.g., 1's complement. When using, e.g., 2's complement, the representation that would otherwise represent the most negative value can be used as a NaN. Especially for unbounded integer types, the inclusion of infinities is advisable, not for overflow, since these do not occur, but in order to have a smallest and a largest value in the type.

B.5.1.1 The integer *result* and *wrap* helper functions

The $result_I$ helper function notifies overflow when the result cannot be represented in I. When an overflow occurs, and recording of indicators is the method for handling (integer) overflows, a continuation value must be given. For bounded integer datatypes, $maxint_F$ and $minint_F$ can be suitable continuation values, if infinities are not representable. In some instances a wrapped result, see below, may be used as continuation value on overflow. Few integer datatypes offer representations for positive and negative infinity. In case such representations are offered, they can be used as continuation values on overflow, similar to their use in floating point datatypes. LIA does not specify the continuation value in this case, that is left to bindings or implementations, but LIA does require that the continuation value(s) be documented.

The $wrap_I$ helper function wraps the result into a value that can be represented in I. The result is wrapped in such a way that the value returned can be used to implement extended range integer arithmetic.

B.5.1.2 Integer maximum and minimum

The operations for integer maximum and minimum are trivial, except taking the maximum or minimum of an empty sequence (empty array, empty list, zero number of parameters, or similar). The case for zero number of parameters is often syntactically excluded (as in Fortran, Common Lisp, and ISLisp), while an empty array or empty list given as a single argument must usually be possible to handle at 'runtime'. LIA specifies an **infinitary** notification for this case. **infinitary** is to be interpreted as "exact infinite result from finite operands", in this case an empty list of numbers. The **infinitary** notification is not specified if any of the arguments is an infinity.

If infinity values are required to be available for a particular integer datatype, a binding may require the continuation values specified to be returned without any **infinitary** notification. When the specified continuation value, $\mathbf{+\infty}$ or $\mathbf{-\infty}$, is not available, other suitable continuation values may be used, and if so they must be documented. If the integer datatype is bounded, but without infinities, $maxint_F$ may be used in place of $\mathbf{+\infty}$ and $minint_F$ may be used instead of $\mathbf{-\infty}$.

Infinities as arguments are not specified for these operations, since infinities are rarely available in integer datatypes. However, compare the specification for max and min operations for floating point datatypes (clause 5.2.2).

B.5.1.3 Integer diminish

Integer diminish is sometimes called 'monus'.

B.5.1.4 Integer power and arithmetic shift

The integer arithmetic shift operations can be used to implement integer multiplication and integer division more quickly in special cases (assuming the shift operation is supported by the hardware, and that support is used).

The shift operations shift either 'right' or 'left' depending on the sign of the second argument. 'Right' shift is done with a positive second argument, and 'left' shift with a negative second argument.

Any continuation value used on overflow here must be documented, either by the binding standard or by the implementation.

B.5.1.5 Integer square root

B.5.1.6 Divisibility tests

Even and odd are simple special cases of the divisibility test offered as separately named operations in several programming languages.

B.5.1.7 Integer division (with floor, round, or ceiling) and remainder

When the result of a division between integers is not an integer, but the final result is required to be an integer, the quotient must be rounded. There are several ways of doing this; floor, ceiling, and unbiased round to nearest being the most important. Truncating, rounding towards zero, is often provided, which, however, may introduce subtle program errors. Integer division, and remainder, is often used for grouping into groups of n items, it is natural to put item i into group $divide(i, n)$. If i can be negative, and truncation is used, group 0 will get $2 \cdot n - 1$ items, rather than the desired n.

pad_I returns the *negative* of the remainder after division and ceiling. The reason for this is twofold: 1) for unsigned integer datatypes the remainder is $\leqslant 0$, and would thus often not be representable unless negated, and 2) it is intuitively easier to think of the "places left in the last unfilled group of equi-sized and packed groups" as a positive entity, a padding.

$residue_I$ can overflow only for unsigned integer datatypes ($minint_I = 0$), and does so for too many cases, and negating it does not change this. $residue_I$ should therefore not be provided for unsigned integer datatypes. $residue_I$ rounds in the same way as $residue_F$. $residue_F$ is often referred to as IEEE remainder.

When there is no exception, for $n \in \mathcal{Z}$ these operations fulfill:

$quot_I(x + n \cdot y, y) = quot_I(x, y) + n$,
$ratio_I(x + 2 \cdot n \cdot y, y) = ratio_I(x, y) + 2 \cdot n$,
$group_I(x + n \cdot y, y) = group_I(x, y) + n$,
$mod_I(x + n \cdot y, y) = mod_I(x, y)$,
$residue_I(x + 2 \cdot n \cdot y, y) = residue_I(x, y)$, and
$pad_I(x + n \cdot y, y) = pad_I(x, y)$.

Note that the div^t_I and rem^t_I from LIA-1 do not fulfill similar useful equalities, due to the disruption around 0 for this pair of operations.

And, when there is no exception, the sign rules are:

$quot_I(x, y) = -group_I(-x, y)$,
$quot_I(x, y) = -group_I(x, -y)$,
$ratio_I(x, y) = -ratio_I(-x, y)$,
$ratio_I(x, y) = -ratio_I(x, -y)$,
$mod_I(x, y) = -pad_I(x, -y)$, and
$residue_I(x, y) = residue_I(x, -y)$.

Finally, when there is no exception, the integer division and remainder operations come in pairs that fulfill:

$quot_I(x, y) \cdot y + mod_I(x, y) = x$,
$ratio_I(x, y) \cdot y + residue_I(x, y) = x$, and
$group_I(x, y) \cdot y - pad_I(x, y) = x$.

B.5.1.8 Greatest common divisor and least common positive multiple

The greatest common divisor is useful in reducing a rational number to its lowest terms. The least common multiple is useful in converting two rational numbers to have the same denominator.

Returning 0 for $gcd_I(0, 0)$, as is sometimes suggested, would be incorrect, since the greatest common divisor for 0 and 0 should be the supremum (upper limit) of $\mathcal{Z}^+$, since elements of $\mathcal{Z}^+$ divide 0. The supremum of $\mathcal{Z}^+$ is infinity. Note also that for an $n > 0$, $gcd_I(n, \mathbf{+\infty})$ should be n, given a reasonable extension of gcd_I to cover infinity arguments.

gcd_I will overflow only if $bounded_I = \mathbf{true}$, $minint_I = -maxint_I - 1$, and both arguments are $minint_I$. The greatest common divisor is then $-minint_I$, which then is not in I.

Least common positive multiple, $lcm_I(x, y)$, overflows for many "large" arguments. E.g., if x and y are relatively prime, then the least common multiple is $|x \cdot y|$, which may be greater than $maxint_I$.

B.5.1.9 Support operations for extended integer range

These operations would typically be used to extend the range of the highest level integer datatype supported by the underlying hardware of an implementation.

The two parts of an integer product, $mul\_ov_I(x, y)$ and $mul\_wrap_I(x, y)$ together provide the complete integer product. Similarly for addition and subtraction. The use of $wrap_I$ guarantees that **overflow** will not occur.

B.5.2 Basic floating point operations

F must be a subset of $\mathcal{R}$. Floating point datatypes can have infinity values as well as NaN values, and also may have a $\mathbf{-0}$. These values are not in F. The special values are, however, commonly available in floating point datatypes today, thanks to the wide adoption of IEC 60599.

Note that for some operations the exceptional value **invalid** is produced only for argument values involving $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, or **sNaN**. For these operations the signature given in LIA-2 does not contain **invalid**.

The report *Floating-Point C Extensions* [57] discusses possible ways of exploiting the IEC 60559 special values, much of which is now integrated in C. The report identifies some of its suggestions as controversial and cites *Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit* [52] as justification.

The following paragraphs is a short summary of the specifications of IEC 60559 regarding the creation and propagation of signed zeros, infinities, and NaNs. There is also some discussion of the material in references [52, 53, 50] where suggestions on this matter are made.

IEC 60559 specifies that 0 and $\mathbf{-0}$ compare equal. The sign is supposed to indicate the direction of approach to zero. The sign is reliable for a zero generated by underflow in a multiplication or division operation, and should be reliable also for operations that approximate elementary transcendental functions (see the LIA-2 specifications in clause 5.3). It is not reliable for a zero generated by an implied subtraction of two floating point numbers with the same value, for which case the zero is arbitrarily given a + sign. The phrase "implied subtraction" indicates either the addition of two oppositely signed numbers or the subtraction of two like signed numbers.

On occurrence of floating overflow or division of a non-zero number by zero, an implementation conforming to IEC 60559 sets the appropriate status flag (if trapping is not enabled) and then continues execution with a result of $\mathbf{+\infty}$ or $\mathbf{-\infty}$ if rounding is to nearest. Infinities as such do *not* indicate that an overflow or division by zero has occurred; infinities can be exact values. IEC 60559 states that the arithmetic of infinities is that associated with mathematical infinities. Thus, an infinity times, plus, minus, or divided by a non-zero finite floating point number yields an infinity for the result; no status flag is set and execution continues. These rules are not necessarily valid for infinities generated by overflow, though they are valid if the infinitary arguments are exact.

NaNs are generated by invalid operations on infinities, 0 divided by 0, and the square root of a negative number (other than $\mathbf{-0}$). Thus NaNs can represent unknown real or complex values, as well as totally undefined values. IEC 60559 requires that the result of any of its basic operations with one or more NaN arguments shall be a NaN. This principle is not extended to the numerical functions by *Floating-Point C Extensions* [57]. The controversial specifications in *Floating-Point C Extensions* [57], *Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit* [52], and *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic* [53] are based on an assumption that all NaN operands represent finite non-zero real-valued numbers.

The LIA-2 policy (for clauses 5.2 and 5.3) for dealing with signed zeros, infinities, and NaNs is as follows:

a) The output is a quiet NaN for any operation for which one (or more) arguments is a quiet NaN, and none of the other arguments is a signalling NaN. There is then no notification.

b) If a mathematical function $h(x)$ is such that $h(0) = 0$, the corresponding operation $op_F(x)$ returns x if $x \in \{0, \mathbf{-0}\}$ and h has a positive derivative at 0, and $op_F(x)$ returns $neg_F(x)$ if $x \in \{0, \mathbf{-0}\}$ and h has a negative derivative at 0.

c) For an argument vector, $\overrightarrow{x}$, where that argument vector involves 0, $\mathbf{-0}$, $\mathbf{+\infty}$, or $\mathbf{-\infty}$, the result of the operation $op_F(\overrightarrow{x})$ is

$$\lim_{\overrightarrow{z} \to \overrightarrow{x}} h(\overrightarrow{z})$$

where an approach to zero is from the positive side if $\overrightarrow{x} = (..., 0, ...)$, and the approach is from the negative side if $\overrightarrow{x} = (..., \mathbf{-0}, ...)$. There is no notification if the limit exists, is finite, and is path independent. The returned value is $\mathbf{+\infty}$ or $\mathbf{-\infty}$ if the limiting value is unbounded, and the approach is towards a point infinitely far from the origin. The returned value is **infinitary**($\mathbf{+\infty}$) or **infinitary**($\mathbf{-\infty}$) if the limiting value is unbounded, and the approach is towards a finite point. The result is **−0** if the limit is zero and the approaching values are path independently negative. The result is 0 if the limit is zero and the approaching values are path independently positive. If a path independent limit does not exist the value returned is **invalid**, and a notification occurs, with a continuation value of **qNaN** if appropriate.

An exception is made for the arc_F and $arcu_F$ operations, where it is found significantly more useful to return certain non-exceptional values for the origin and for the four double infinity argument cases, than to return an exceptional value, even with non-NaN continuation values.

B.5.2.1 The rounding and floating point *result* helper functions

The $result_F$ helper function notifies overflow when the result is too large to be approximated by a value in F. The $result_F$ helper function notifies underflow when there is (risk for) denormalisation loss for a tiny result. The $result_F$ helper function also ensures that a properly signed zero is the continuation value when a zero is appropriate for an underflow continuation value. When an overflow or underflow occurs, and recording of indicators is the method for handling (floating point) overflow or underflow, a continuation value must be provided. LIA-2 specifies a continuation value, and if that can be represented in the target datatype, that value should be used as continuation value. If the parameter iec_559_F has the value **true**, then IEC 60559 in many cases require particular continuation values (consistent with what is specified by LIA-2) to be used.

The continuation values for overflow are defined to be in accordance with IEC 60559. These particular choices for continuation values are useful for interval arithmetic.

B.5.2.2 Floating point maximum and minimum

As for the integer case, the maximum and minimum of empty sequences need be handled, but for floating point datatypes, infinities are usually available.

For floating point datatypes there is also usually a negative zero available, and returning the correct sign on a zero result for the maximum and minimum operations requires more than simple comparisons to implement. The signs of zeroes may need to be inspected using *copysign* or *isnegativezero*.

B.5.2.3 Floating point diminish

As for the integer case, this operation computes the positive difference. Note that $dim_F(+\infty, +\infty) =$ **invalid**(**qNaN**) is consistent with that $sub_F(+\infty, +\infty) =$ **invalid**(**qNaN**) according to IEC 60559.

An implementation of dim_F could be `if x >= y then x-y else 0`.

B.5.2.4 Floor, round, and ceiling

Since $fmax_F$ always has an integral value for floating point types that conform to LIA-1, no overflow can occur for these operations.

Note that the sign of a zero result is maintained in accordance with IEC 60559:

$$\begin{aligned}&\mathit{floor}_F(x) = \mathit{neg}_F(\mathit{ceiling}_F(\mathit{neg}_F(x)))\\&\mathit{rounding}_F(x) = \mathit{neg}_F(\mathit{rounding}_F(\mathit{neg}_F(x)))\\&\mathit{ceiling}_F(x) = \mathit{neg}_F(\mathit{floor}_F(\mathit{neg}_F(x)))\end{aligned}$$

Negative zeroes, if available, are handled in such a way as to maintain these identities.

Note that $\mathit{rounding\_rest}_F$ always is an exact operation, while $\mathit{floor\_rest}_F$ is not always exact for negative arguments, and $\mathit{ceiling\_rest}_F$ is not always exact for positive arguments.

B.5.2.5 Remainder after division and round to integer

The remainder after division and unbiased round to integer (IEC 60559 remainder, or IEEE remainder) is always an exact operation (unless there is an implied division by zero), even if the floating point datatype only conforms to LIA-1, but not to the more specific IEC 60559.

Remainder after floating point division and floor to integer cannot be exact for all pairs of arguments from F. For a small negative numerator and a positive denominator, the resulting value loses much absolute accuracy in relation to the original value. Such an operation is therefore not included in LIA-2. Similarly for floating point division and ceiling.

See also the radian normalisation and the argument angular-unit normalisation operations (5.3.8.1, and especially 5.3.9.1).

B.5.2.6 Square root and reciprocal square root

The inverses of squares are double valued, the two possible results having the same magnitude with opposite signs. For a non-zero result, LIA-2 requires that each of the corresponding operations return a positive result.

$\sqrt{x}$ cannot be exactly halfway between two values in F if $x \in F$. For $\sqrt{x}$ to be exactly halfway between two values in F would require that it had exactly $(p + 1)$ digits (last digit non-zero) for its exact representation. The square of such a number would require at least $(2 \cdot p + 1)$ digits with last $p + 1$ digits not all zero, which could not equal the p-digit number x.

The extensions $\mathit{sqrt}_F(+\infty) = +\infty$ and $\mathit{sqrt}_F(\mathbf{-0}) = \mathbf{-0}$ are mandated by IEC 60559. LIA-2 also requires that these hold for implementations which support infinities and signed zeros. However, it should be noted that while the second is harmless, the first may lead to erroneous results for a $+\infty$ generated by an addition or subtraction with result just barely outside of $[-\mathit{fmax}_F, \mathit{fmax}_F]$ after rounding. Hence its square root would be well within the representable range. The possibility that LIA-2 should require that $\mathit{sqrt}_F(+\infty) = \textbf{invalid}(+\infty)$ was considered, but rejected because of the principle of regarding arguments as exact, even if they are not exact, when there is a non-degenerate neighbourhood around the argument point, for which the mathematical function on $\mathcal{R}$ is defined. In addition $\mathit{sqrt}_F(+\infty) = +\infty$ is already required by IEC 60559.

Note that the requirement that $\mathit{sqrt}_F(x) = \textbf{invalid}(\textbf{qNaN})$ for x strictly less than zero is mandated by IEC 60559. It follows that NaNs generated in this way represent imaginary values, which would become complex through addition and subtraction, and even imaginary infinities on multiplication by ordinary infinities.

The $\mathit{rec\_sqrt}_F$ operation will increase performance for scaling a vector into a unit vector. Such an operation involves division of each component of the vector by the magnitude of the vector or, equivalently and with higher performance, multiplication by the reciprocal of the magnitude.

B.5.2.7 Multiplication to higher precision floating point datatype

This operation is intended for the case that there exist at least two floating point datatypes F and F', ideally such that the product of two numbers of type F is always exactly representable in type F'.

To obtain higher precision for multiplication, in the absence of a suitable level of precision F', a programmer can exploit the paired mul_F and mul_lo_F operations.

B.5.2.8 Support operations for extended floating point precision

These operations would typically be used to extend the precision of the highest level floating point datatype supported by the underlying hardware of an implementation. There is, however, no intent to provide a set of operations suitable for the implementation of a *complete* package for the support of calculations at an arbitrarily high level of precision.

The major motivation for including them in LIA-2 is to provide a capability for accurately evaluating residuals in an iterative algorithm. The residuals give a measure of the error in the current solution. More importantly they can be used to estimate a correction to the current solution. The accuracy of the correction depends on the accuracy of the residuals. The residuals are calculated as a difference in which the number of leading digits cancelled increases as the accuracy of the solution increases. A doubled precision calculation of the residuals is usually adequate to produce a reasonably efficient iteration.

For the basic floating point arithmetic doubled precision operations, the high parts may be calculated by the corresponding floating point operations as specified in LIA-1. Note, however, that in order to implement exact floating point addition and subtraction, rnd_F must round to nearest. If $add_F(x, y)$ rounds to nearest then the high and low parts represent $x + y$ exactly.

When the high parts of an addition or subtraction overflows, the low parts, as specified by LIA-2, return their results as if there was no overflow. add_lo_F and sub_lo_F can underflow only when subnormals are not supported. In addition, if the high part underflows, then the low part is zero.

The product of two numbers, each with p digits of precision, is always exactly representable in at most $2 \cdot p$ digits. The high and low parts of the product will always represent the true product.

The remainder for division is more useful than a $2 \cdot p$-digit approximation. The remainder will be exactly representable if the high part differs from the true quotient by less than one *ulp*. The true quotient can be constructed p digits at a time by division of the successive remainders by the divisor.

The remainder for square root is more useful than a low part for the same reason that the remainder is more useful for division. The remainder for the square root operation will be exactly representable only if the high part is correctly rounded to nearest, as is required by the specification for $sqrt_F$.

See *Semantics for Exact Floating Point Operations* [62] for more information on exact floating point operations.

See *Proposal for Accurate Floating-Point Vector Arithmetic* [63] for more information on exact, or high accuracy, floating point summation and dot product. These operations may be the subject of an amendment to LIA-2.

B.5.3 Elementary transcendental floating point operations

The basic floating point operations of LIA-2 and the elementary transcendental floating point operations have been separated into two different clauses of LIA-2, since they use slightly different specification mechanisms. The basic floating point operations need no approximation helper functions. The elementary transcendental floating point operations need approximation helper function in order to express the wider error tolerance for these operations.

B.5.3.1 Maximum error requirements

The $max\_error\_op_F$ parameters measure the discrepancy between the computed value $op_F(x)$ and the true mathematical value $f(x)$ in ulps of the true value. The magnitude of the error bound is thus available to a program from the computed value $op_F(x)$. Note that for results at an exponent boundary for F, y, the error away from zero is in terms of $ulp_F(y)$, whereas the error toward zero is in terms of $ulp_F(y)/r_F$, which is the ulp of values slightly smaller in magnitude than y.

Within limits, accuracy and performance may be varied to best meet customer needs. Note also that LIA-2 does not prevent a vendor from offering two or more implementations of the various operations.

The operation specifications define the domain and range for the operations. This is done partly by the given signature, and partly by the specification of cases that do not return **invalid**. In addition, the computational domain and range are more limited for the operations than for the corresponding mathematical functions because the arithmetic datatypes are subsets of $\mathcal{R}$. Further, any (conforming) F is limited in range, and the operations may return an overflow or an underflow. Thus the actual domain of $exp_F(x)$ is approximately given by $x \leqslant \ln(fmax_F)$. For larger values of x, $exp_F(x)$ will overflow, though for $x = +\infty$ the exact result $+\infty$ will be returned. The actual range extends over positive F, although there are non-negative values, $v \in F$, for which there is no $x \in F$ satisfying $exp_F(x) = v$.

The thresholds for the **overflow** and **underflow** notifications are determined by the parameters defining the arithmetic datatypes. The threshold for an **invalid** notification is determined by the domain of arguments for which the mathematical function being approximated is defined. The **infinitary** notification is the operation's counterpart of a mathematical pole of the mathematical function being approximated by the operation. The threshold for **absolute_precision_underflow** is determined by the parameters $big\_angle\_r_F$ and $big\_angle\_u_F$.

LIA-2 imposes a fairly tight bound on the maximum error allowed in the implementation of each operation. The tightest possible bound is given by requiring rounding to nearest, for which the accompanying performance penalty is often unacceptably high for the operations approximating elementary transcendental functions. LIA-2 does not require round to nearest for such operations, but allows for a slightly wider error bound characterised via the $max\_error\_op_F$ parameters. The $max\_error\_op_F$ parameters must be documented by the implementation for each such parameter required by LIA-2. A comparison of the values of these parameters with the values of the specified maximum value for each such parameter will give some indication of the "quality" of the routines provided. Further, a comparison of the values of this parameter for two versions of a frequently used operation will give some indication of the accuracy sacrifice made in order to gain performance.

Language bindings are free to modify the error limits provided in the specifications for the operations to meet the expected requirements of their users.

Material on the implementation of high accuracy operations is provided in for example [50, 52, 59].

B.5.3.2 Sign requirements

The requirements imply that the sign of the result or continuation value is to be reliable, except for the sign of an infinite result or continuation value, where except for a signed zero argument, it is often the case that one cannot determine the sign of the infinity. Still for sign symmetric mathematical functions, the approximating operation is also sign symmetric, including infinitary results.

Note that the sign requirements stated generally imply some requirements that are not given explicitly for each operation specification in LIA-2. For example, $sin^*_F(n \cdot 2 \cdot \pi + \pi) = 0$ is a requirement implied by the general sign requirements.

B.5.3.3 Monotonicity requirements

A maximum error of 0.5 ulp implies that an approximation helper function must be a monotonic approximation to the mathematical function. When the maximum error is greater than 0.5 ulp, and the rounding is not directed, this is not automatically the case.

There is no general requirement that the approximation helper functions are *strictly* monotone on the same intervals on which the corresponding exact function is strictly monotone, however, since such a requirement cannot be made due to the fact that all floating point types are discrete, not continuous.

The monotonicity requirements are not extended to the angular unit arguments (for the operations that take such an argument or arguments). The reason for this is that it is thought both hard to implement, and also of no interest to users to have monotonicity on that (those) argument(s), since the angular unit is not normally varied, except when converting between angular units, and even then the unit arguments involved are usually constants.

The monotonicity requirements together with the extra accuracy requirements also imply requirements not explicitly stated. For example $-1 \leqslant sin^*_F(x) \leqslant 1$ is such an implied requirement. Therefore, even if some of the extra accuracy requirements are relaxed (see annex A), it may be necessary to reintroduce some of the requirements that were implied.

B.5.3.4 The $result^*$ helper function

The $result^*_F$ helper function is more suitable than the $result_F$ helper function when the approximation is not guaranteed to be 0.5 ulp nor guaranteed to be directed.

Ideally, however, though not expressed in the LIA-2 specifications, also the operations approximating elementary transcendental functions obey the rounding mode (if the implementation has rounding modes) in the sense that "round towards negative infinity" gives a result that is less than or equal to the true result (and similarly for "round towards positive infinity"). However, and in contrast to the basic arithmetic operations, the error may then be more than 1 ulp. So even if the rounding modes are heeded also for the operations approximating elementary transcendental functions, the interpretation of the rounding modes are still looser than for the basic arithmetic operations. LIA-2 as such does not require even this looser interpretation. It is up to implementations, and the accompanying documentation, to implement this, if desired, and to document the behaviour of these operations under different rounding modes. To get reliable upper and lower bounds of the true result, that are also close to the true result, is useful for interval arithmetic. Still, using the returned result, within the error bounds specified by LIA-2, together with the relevant $max\_error\_op_F$ parameter, one can get a (perhaps slightly wider) safe interval containing the true result.

B.5.3.5 Hypotenuse

The $hypot_F$ operation can produce an overflow only if both arguments have magnitudes very close to the overflow threshold. $hypot_F$ only underflows if both arguments are subnormal numbers. Care must be taken in its implementation to either avoid or properly handle overflows and underflows which might occur in squaring the arguments. The function approximated by this operation is mathematically equivalent to complex absolute value, which is needed in the calculation of the argument (also called phase; see arc_F) and modulus (also called absolute value; $hypot_F$) of a complex number. It is important for this application that an implementation satisfy the constraint on the magnitude of the result returned.

LIA-2's $hypot_F$ does not follow the recommendations in *Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit* [52] and in *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic* [53] which recommend that

$$hhypot_F(+\infty, \mathbf{qNaN}) = +\infty$$
$$hhypot_F(-\infty, \mathbf{qNaN}) = +\infty$$
$$hhypot_F(\mathbf{qNaN}, +\infty) = +\infty$$
$$hhypot_F(\mathbf{qNaN}, -\infty) = +\infty$$

which are based on the claim that a **qNaN** represents an (unknown) real valued number. Such a claim is not always valid, though it may sometimes be.

B.5.3.6 Operations for exponentiations and logarithms

For all of the exponentiation operations, overflow occurs for sufficiently large values of the argument(s).

There is a problem for $power_F(x, y)$ if both x and y are zero:

– Ada raises an 'exception' for the operation that is close in semantics to $power_F$ when both arguments are zero, in accordance with the fact that 0^0 is mathematically undefined.

– The X/OPEN Portability Guide, as well as C99, specifies for `pow(0.0, 0.0)` a return value of 1, and no notification. Those specifications agree with the recommendations in [50, 52, 53, 56].

The specification in LIA-2 follows Ada, and returns **invalid** for $power_F(0, 0)$, because of the risks inherent in returning a result which might be inappropriate for the application at hand. Note however, that $power_{F,I}(0, 0)$ is 1, without any notification. The reason is that the limiting value for the corresponding mathematical function, when following either of the only two continuous paths, is 1. This also agrees with the Ada specification for a floating point value raised to a power in an integer datatype, as well as that for other programming languages which distinguish these operations. The C99 (and X/OPEN) specification for the `pow` can be regarded as a combination of the $power_{F,I}$ and $power_F$ operations. Due to this combination, LIA-2 has a requirement that

$power_F$ for integral second arguments and $power_{F,I}$ (in the same library) are related by equality for positive first arguments.

Along any path defined by $y = k/\ln(x)$ the mathematical function x^y has the value e^k. It follows that some of the limiting values for x^y depend on the choice of k, and hence are undefined, as indicated in the specification.

The result of the $power_F$ operation is **invalid** for negative values of the base x. The reason is that the floating point exponent y might imply an implicit extraction of an even root of x, which would have a complex value for negative x. This constraint is explicit in Ada, and is widely imposed in existing numerical packages provided by vendors, as well as several other programming languages.

The arguments of $power_F$ are floating point numbers. No special treatment is provided for integer floating point values, which may be approximate. The cases for integer values of the arguments are covered by the operations $power_{F,I}$ and $power_I$. In the example binding for C a specification for pow_F is supplied. pow_F combines $power_F$ and $power_{F\mathcal{Z}}$ in a way suitable for C's `pow` operation.

For implementations of the $power_F$ operation there is an accuracy problem with an algorithm based on the following, mathematically valid, identity:

$$x^y = r_F^{y \cdot \log_{r_F}(x)}$$

The integer part (floor plus one, *not* truncation) of the product $y \cdot \log_{r_F}(x)$ defines the exponent of the result and the remaining fractional part defines the reduced argument. If the exponent is large, and one calculates p_F digits of this intermediate result, there will be fewer than p_F digits for the fraction. Thus, in order to obtain a reduced argument accurately rounded to p_F digits, it may be necessary to calculate an approximation to $y \cdot \log_{r_F}(x)$ to a few more than $\log_{r_F}(emax_F) + p_F$ base r_F digits.

In Ada95 the operation closest to $power_{F,I}$ is specified to be computed by successive multiplications, for which the error in the evaluation increases linearly with the size of the exponent. In a strict Ada implementation there is no way that a prescribed error limit of a few ulps can be met for large exponents.

The special exponentiation operations, corresponding to 2^x and 10^x, have specifications which are minor variations on those for $exp_F(x)$. Accuracy and performance can be increased if they are specially coded, rather than evaluated as, e.g., $exp_F(mul_F(x, ln_F(2)))$ or $power_F(2, x)$. Similar comments hold for the base 2 and base 10 logarithms operations.

The $expm1_F$ operation has two advantages: Firstly, $expm1_F(x)$ is much more accurate than $sub_F(exp_F(x), 1)$ when the exponent argument is close to zero. Secondly, the $expm1_F$ operation does not **underflow** for "very" negative exponent arguments, something which may be advantageous if **underflow** handling is slow, and high accuracy for "very" negative arguments is not needed. Note in addition that underflow is avoided for this operation. This can be done only since LIA-2 adds requirements beyond those of LIA-1 regarding minimum precision (see clause 4). If those extra requirements were not done, underflow would not be justifiably removable for this operation. Similar argumentation applies to $ln1p_F$.

Similarly, there are two advantages with the $power1pm1_F$ operation: Firstly, $power1pm1_F(b, x)$ is much more accurate than $sub_F(power_F(add_F(1, b), x), 1)$ when the exponent argument is close to zero. Secondly, the $power1pm1_F$ operation does not **underflow** for "very" negative exponent arguments (when the base is greater than 1), something which may be advantageous if **underflow** handling is slow, and high accuracy for "very" negative arguments is not needed.

The handling of infinites and negative zero as arguments to the exponentiation and logarithm operations, like for all other LIA operations, follow the principles for dealing with these values as explained in section B.5.2. Note in particular that $logbase_F(b, x)$ is specified so as to be consistent with $div_F(ln_F(x), ln_F(b))$ except that $logbase_F(b, x)$ is required to be more accurate.

The $expm1_F$ and $ln1p_F$ operations are required to return the argument when the argument is in a certain interval around 0. Some floating point parameters from LIA-1 had to be made a bit stricter for LIA-2 to guarantee that this interval always is wider than the interval of subnormal numbers (this change is to be integrated with LIA-1 when LIA-1 is revised). This way underflow can always be avoided for these operations, and in the interval specified, they can with high accuracy return the argument unchanged.

Several of the operations have requirements that push the result towards a finite limiting value, so that that the limiting value is actually reached (within a reasonable margin) *after* rounding, even if the limiting value cannot, or otherwise need not, be reached before rounding. Similar requirements appear also in the other subclauses of clause 5.3.

Note also that even the use of the nearest approximation to e that is representable in F as a base argument to the $power_F$ and $logbase_F$ operations do not produce a duplication of exp_F and ln_F.

B.5.3.7 Introduction to operations for trigonometric elementary functions

The real trigonometric functions sin, cos, tan, cot, sec, and csc are all periodic. The smallest period for sin, cos, sec, and csc is $2 \cdot \pi$ radians (360 degrees). The smallest period for tan and cot is π radians (180 degrees) (and thus also have a period of $2 \cdot \pi$ radians (360 degrees)). The mathematical trigonometric functions are perfectly periodic. Their numerical counterparts are not that perfect, for two reasons.

Firstly, the radian normalisation cannot be exact, even though it can be made very good given very many digits for the approximation(s) of π used in the angle normalisation, returning an offset from the nearest axis, and including guard digits. The unit argument normalisation, however, can be made exact regardless of the (non-zero and, in case $denorm_F =$ **false** not too small) unit and the original angle, returning only a plain angle in F. LIA-2 requires unit argument angle normalisation to be exact.

Secondly, the length of one revolution is of course constant, but the density of floating point values gets sparser (in absolute spacing rather than relative) the larger the magnitude of the values are. This means that the number of floating point values gets sparser per revolution the larger the magnitude of the angle value is. For this reason the notification **absolute_precision_underflow** is introduced, together with two parameters, one for radians and one for other angular units. This notification is given when the magnitude of the angle value is “too big”. Exactly when the representable angle values get too sparse depends upon the application at hand, but LIA-2 gives a default value for the parameters that define the cut-off.

Note that the **absolute_precision_underflow** notification is unrelated to any argument reduction problems. Argument reduction is (implicitly for radians, explicitly for other angular units) required by LIA-2 to be very accurate. But no matter how accurate the argument reduction is, floating point values are still sparser in absolute terms the larger the values are. The trigonometric operations return a result within about an ulp, and that high accuracy is wasted if the angular argument is not kept at a high accuracy too, both relative and absolute.

LIA-2 includes specifications for high accuracy angle normalisation operations, both for radians and for other angular units. The angle normalisation operations give a result within minus half a cycle to plus half a cycle (as does the angle conversion operations), unless the argument angular value is too big (or there is some other error). These operations should be used to keep the representation of angles at a high accuracy. LIA-2 also includes angle normalisation operations that can be used to maintain an even higher degree of accuracy, giving the offset from the nearest axis (though without any extra guard digits). To use these, one need to keep track of the currently nearest axis, and make appropriate adjustments in the calculations, which unfortunately complicates programs that use these nearest-axis normalisations.

Note that $rad(x) = \arccos(\cos(x))$ if $\sin(x) > 0$ and $rad(x) = -\arccos(\cos(x))$ if $\sin(x) < 0$. The first part of $axis\_rad(x)$ indicates which axis is nearest to the angle x. The second part of $axis\_rad(x)$ is an angle offset from the axis that is nearest to the angle x. The second part of $axis\_rad(x)$ is equal to $rad(x)$ if $\cos(x) \geqslant 1/\sqrt{2}$ (i.e. if the first part of $axis\_rad(x)$ is $(1, 0)$). More generally, the second part of $axis\_rad(x)$ is equal to $rad(4 \cdot x)/4$.

$rad(x)$ returns the same angle as the angle value x, but the returned angle value is between $-\pi$ and π. The *rad* function is defined to be used as the basis for the angle normalisation operations. The *axis_rad* function is defined to be used as the basis for a numerically more accurate radian angle normalisation operation. The *arc* function is defined to be used as the basis for the arcus (angle) operations, which are used for conversion from Cartesian to polar co-ordinates.

B.5.3.8 Operations for radian trigonometric elementary functions

The radian trigonometric approximation helper functions (including those for normalisation and conversion from radians) are required to have the same zero points as the approximated mathematical function only if the absolute value of the argument is less than or equal to $big\_angle\_r_F$. Likewise, the radian trigonometric approximation helper functions are required to have the same sign as the approximated mathematical function only if the absolute value of the argument is less than or equal to $big\_angle\_r_F$. Indeed, the radian trigonometric approximation helper functions need not be defined at all outside of $[-big\_angle\_r_F, big\_angle\_r_F]$.

The $big\_angle\_r_F$ parameter may be adjusted by bindings, or even by some compiler flag, or mode setting within a program. However, this method should only allow the value of this parameter to be set to a value greater than $2 \cdot \pi$, so that at least arguments within the first two (plus and minus) cycles are allowed, and such that $ulp_F(big\_angle\_r_F) < \pi/1000$, so that at least 2000 evenly distributed points within the 'last' cycle (farthest away from 0) are distinguishable. The latter gives a rather low accuracy at the far ends of the range, especially if p_F is comparatively large, so values this large for $big\_angle\_r_F$ are not recommendable unless the application is such that high accuracy trigonometric operations are not needed. Note that if $big\_angle\_r_F$ is allowed to be increased, then, for conformity with LIA-2, the radian angle reduction may need to be made more precise.

For reduction of an argument given in radians, implementations use one or several approximate value(s) of π (or of a multiple of π), valid to, say, n digits. The division implied in the argument reduction cannot be valid to more than n digits, which implies a maximum absolute angle value for which the reduction yields an accurate reduced angle value.

Regarding argument reduction for radians, there is a particular problem when the result of the trigonometric operation is very small (or very big), but the angular argument is not very small. In such cases the argument reduction must be very accurate, using an extra-precise approximation to π, relative to what is normally used for arguments of similar magnitude, so that significant digits in the result are not lost. Such loss would imply non-conformance to LIA-2 by the error in the final result being greater than that specified by LIA-2. In general, extra care has to be taken when the second part of $axis\_rad(x)$ is close to 0.

Note that

– tan and sec have poles at odd multiples of $\pi/2$ radians (90 degrees).

– cot and csc have poles at multiples of π radians (180 degrees).

All four of the corresponding operations with poles may produce **overflow** for arguments sufficiently close to the poles of the functions. The tan_F operation produces no **infinitary** notification. The reason is that the poles of $\tan(x)$ are at odd multiples of $\pi/2$, which are not representable in F. The mathematical cotangent function has poles at even multiples of $\pi/2$, of which only the origin is representable in F. For a system which supports signed zeros and infinities, the continuation values are $\mathbf{+\infty}$ and $\mathbf{-\infty}$ for arguments of 0 and $\mathbf{-0}$ respectively to $cot_F(x)$. Although the mathematical function sec has poles at odd multiples of $\pi/2$, the sec_F operation will not generate any **infinitary** notification because such arguments are not representable in F.

The **infinitary** notification cannot occur for any non-zero argument in radians because π is not representable in F, nor is $\pi/2$. For the angular unit argument trigonometric operations the sign of the infinitary continuation value has been chosen arbitrarily for an **infinitary** which occurs for a non-zero argument. However, sign symmetry, when appropriate, is maintained.

The operations may produce **underflow** for arguments sufficiently close to the zeros of the function. For a subnormal argument x, the sin_F, tan_F, $arcsin_F$, and $arctan_F$ return x for the result, with very high accuracy. Similarly, for a subnormal argument, cos_F and sec_F can return a result of 1.0 with very high accuracy.

The trigonometric inverses are multiple valued. They are rendered single valued by defining a principal value range. This range is closely related to a branch cut in the complex plane for the corresponding complex function. Among the floating point numerical functions this branch cut is “visible” only for the arc_F operation. The arc function has a branch cut along the negative real axis. For $x < 0$ the function has a discontinuity from $-\pi$ to $+\pi$ as y passes through zero from negative to positive values. Thus for $x < 0$, systems supporting signed zeros can handle the discontinuity as follows:

$$arc_F(x, \mathbf{-0}) = up_F(-\pi)$$
$$arc_F(x, 0) = down_F(\pi)$$

There is a problem for zero argument values for this operation. The values given for the operation $arc_F(x, y)$ for the four combinations of signed zeros for x and y are those given in [52]. The following table of values is given in [52] for the value of $arc_F(x, y)$ with both of the arguments zero:

Zero arguments		
x	y	$arc_F(x, y)$
0	0	0
$\mathbf{-0}$	0	π
$\mathbf{-0}$	$\mathbf{-0}$	$-\pi$
0	$\mathbf{-0}$	$\mathbf{-0}$

Note that the mathematical arc function is indeterminate (undefined) for (0,0), but close representable approximations the above result are numerically more useful than giving an **invalid** notification for such arguments. LIA-2 therefore specifies results as above.

There is also a problem for argument values of **+∞** or **−∞** for this operation. The following table of values is given in [52] for the value of $arc_F(x, y)$ with at least one of the arguments infinite:

Infinite arguments		
x	y	$arc_F(x, y)$
+∞	$\geqslant 0$	0
+∞	**+∞**	$\pi/4$
finite	**+∞**	$\pi/2$
−∞	**+∞**	$3 \cdot \pi/4$
−∞	$\geqslant 0$	π
−∞	**−0**	$-\pi$
−∞	< 0	$-\pi$
−∞	**−∞**	$-3 \cdot \pi/4$
finite	**−∞**	$-\pi/2$
+∞	**−∞**	$-\pi/4$
+∞	< 0	**−0**
+∞	**−0**	**−0**

If one of x and y is infinite and the other is finite, the result tabulated is consistent with that obtained by a conventional limiting process. However, the results of $\pi/4$, $-\pi/4$, $3 \cdot \pi/4$, and $-3 \cdot \pi/4$ corresponding to infinite values for both x and y, are of questionable validity, since only the quadrant is known, not the angle within the quadrant. However, these results are numerically more useful than giving an **invalid** notification for such arguments. LIA-2 therefore specifies results as above.

B.5.3.9 Operations for trigonometrics with given angular unit

At present only Ada specifies trigonometric operations with angular unit argument. LIA-2 has adopted angular unit argument operations in order to encourage uniformity among languages which might include such operations in the future. The angular units in T appear to be particularly important and have therefore been given a tighter error bound requirement. An implementation can of course have the same (tighter) error bound for all angular units. Some programming languages provide trigonometric operations with an implicit angular unit argument with value 360.

The trigonometric approximation helper functions with angular unit argument (including those for normalisation and conversion from radians) are required to have the same zero points as the approximated mathematical function. Likewise, the trigonometric approximation helper functions with angular unit argument are required to have the same sign as the approximated mathematical function. However, the trigonometric approximation helper functions with angular unit argument need not be defined at all outside of $[-big\_angle\_u_F \cdot |u|, big\_angle\_u_F \cdot |u|]$, where u is the value of the angular unit argument.

The $big\_angle\_u_F$ parameter may be adjusted by bindings, or even by some compiler flag, or mode setting within a program. However, this method should only allow the value of this parameter to be set to a value greater than or equal to 1, so that at least arguments within the first two (plus and minus) cycles are allowed, and such that $ulp_F(big\_angle\_u_F) \leqslant 1/2000$, so that at least 2000 evenly distributed points within the 'last' cycle (farthest away from 0) are distinguishable. The latter gives a rather low accuracy at the far ends of the range, especially if p_F is comparatively large, so values this large for $big\_angle\_u_F$ are not recommendable unless the application is such that high accuracy trigonometric operations are not needed.

The $min\_angular\_unit_F$ parameter is specified for two reasons. Firstly, if the type F has no subnormal values ($denorm_F$ = **false**), some angle values in F are not representable after normalisation if the angular unit has too small magnitude. This gives the firm limit given in cluase 5.3.9. Secondly, even if F has subnormal values ($denorm_F$ = **true**), angular units with very small magnitude do not allow the representable angles to be particularly dense, not even if the angular value is within the first cycle. This does in itself not give rise to a particular limit value, but the limit value defined in cluase 5.3.9 is reasonable.

Provided that $|u| \geqslant min\_angular\_unit_F$, an angular unit u can be either positive or negative. If it's negative, growing angular values turns the angle "clockwise" rather than counter-clockwise as for radians and other positive angular units. Ada does not permit negative angular units, but since there is no mathematical nor numerical reason to not allow them, LIA-2 allows negative angular unit argument values, avoiding an unjustifiable and arbitrary decision to disallow them. This only very marginally complicates the specifications given in LIA-2 as well as the implementations that follow those specifications.

Note that the angular unit argument need not be integral, even though several common non-radian angular units are integral, 360, 400, etc. Note also that even the use of the nearest approximation to $2 \cdot \pi$ that is representable in F as angular unit argument does not produce a duplication of the radian trigonometric operations. The radian trigonometric operations need to use one or more approximations to π (or an integer fraction of π) that are more accurate than can be represented in F, in order to fulfill the accuracy requirements of LIA-2.

All of the argument angular unit trigonometric, and argument angular unit inverse trigonometric, approximation helper functions, including those for normalisation, angular unit conversion, and arc, are exempted from the monotonicity requirement for the angular unit argument.

If the angular unit argument, u, is such that $u/4 \in F$, the $tanu_F$ operation has poles at odd multiples of $u/4$. This is the case for degrees ($u = 360$). As for $tanu_F$, if the angular unit argument, u, is such that $u/4 \in F$ the $secu_F$ operation has poles (**infinitary**) at odd multiples of $u/4$.

The same comments hold for the $arcu_F$ operation as for arc_F operation, except that the discontinuity in the mathematical function is from $-u/2$ to $+u/2$.

B.5.3.10 Operations for angular-unit conversions

Angular conversion operations are commonly found on 'scientific' calculators and also in Java, though then often only between degrees and radians.

Conversion of an angular value x from angular unit a to angular unit b appears simple: compute $x \cdot b/a$. Basing a numerical conversion of angular values directly on the above mathematical equality (e.g. $div_F(mul_F(x, b), a)$) loses much absolute angular accuracy, however, especially for large angular values. Instead computing $arcu_F(b, cosu_F(a, x), sinu_F(a, x))$ then gives a more accurate result. This might still not be within the accuracy required by LIA-2 for the angular unit conversion operations specified by LIA-2, which here requires a maximum error of 0.5 ulp.

Note that all of the angular conversion operations return an angularly normalised result. This is in order to maintain high absolute accuracy of the angle being represented.

B.5.3.11 Operations for hyperbolic elementary functions

The hyperbolic sine operation, $sinh_F(x)$, will overflow if $|x|$ is in the immediate neighbourhood of $\ln(2 \cdot fmax_F)$, or greater.

The hyperbolic cosine operation, $cosh_F(x)$, will overflow if $|x|$ is in the immediate neighbourhood of $\ln(2 \cdot fmax_F)$, or greater.

The hyperbolic cotangent operation, $coth_F(x)$, has a pole at $x = 0$.

The inverse of cosh is double valued, the two possible results having the same magnitude with opposite signs. The value returned by $arccosh_F$ is always greater than or equal to 1.

The inverse hyperbolic tangent operation $arctanh_F(x)$ has poles at $x = +1$ and at $x = -1$.

The inverse hyperbolic cotangent operation $arccoth_F(x)$ has poles at $x = +1$ and at $x = -1$.

Like for the exponentiation and logarithm operations, there are extra accuracy requirements, for certain arguments.

When appropriate, there are also sign symmetry requirements on the approximation helper functions. These sign symmetry requirements for “ordinary” arguments are followed through in the operation specification to cover also signed zeroes and infinites. Similar requirements appear also in the other subclauses of clause 5.3.

For $sinh_F$, $tanh_F$, $arcsinh_F$, and $arctanh_F$, for a specified interval around 0, the operation returns its argument unchanged, and does so with high accuracy. Underflow notifications are also avoided for these cases, since there is no denormalisation loss.

B.5.4 Operations for conversion between numeric datatypes

Clause 5.2 of LIA-1 covers conversions from an integer type to another integer type and to a floating point type, as well as between (LIA-1 conforming) floating point types of the same radix.

LIA-2 extends these conversions to cover conversions to and from non-LIA conforming datatypes, such as conversion to and from strings, and also extends the floating point conversion specifications to handle conversions where the radices may be different.

In ordinary string formats for numerals, the string “Hello world!” is an example of a signalling NaN.

LIA-2 does not specify any string formats, not even for the special values $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, and quiet NaN, but possibilities for the special values include the strings used in the text of LIA-2, as well as strings like “+infinity” or “positiva oändligheten”, etc, and the strings used may depend on preference settings, as they may also for non-special values. For instance, one may use different notation for the decimal separator character (like period, comma, Arabic comma, ...), use superscript digits for exponents in scientific notation, or use Arabic digits or traditional Thai digits. String formats for numerical values, and if and how they may depend on preference settings, is also an issue for bindings or programming language specifications, not for this part of LIA.

If the value converted is greater than those representable in the target, or less than those representable in the target, even after rounding, then an overflow will result. E.g., if the target is a character string of at most 3 digits, and the target radix is 10, then an integer source value of 1000 will result in an overflow. As for other operations, if the notification handling is by recording of indicators, a suitable continuation value must be used.

Most language standards contain (partial) format specifications for conversion to and from strings, usually for a decimal representation.

LIA-2 requires, like C99, all floating point conversion operations to be such that the error is at most 0.5 ulp. This is in contrast to IEC 60559, which allows conversion operations to have an error of up to 0.97 ulp.

B.5.5 Numerals as operations in a programming language

B.5.5.1 Numerals for integer datatypes

Negative values (except $minint_I$ if $minint_I = -maxint_I - 1$) can be obtained by using the negation operation (neg_I).

Integer numerals in radix 10 are normally available in programming languages. Other radices may also be available for integer numerals, and the radix used may be part of determining the target integer datatype. E.g., radix 10 may be for signed integer datatypes, and radix 8 or 16 may be for unsigned integer datatypes.

Syntaxes for numerals for different integer datatypes need not be different, nor need they be the same. This part does not further specify the format for integer numerals. That is an issue for bindings.

Overflow for integer numerals can be detected at “compile time”, and warned about. Likewise can notifications about invalid, e.g. for infinitary or NaN numerals that cannot be converted to the target type, be detected at “compile time” and be warned about.

B.5.5.2 Numerals for floating point datatypes

If the numerals used as operations in a program, and numerals read from other sources use the same radix, then “internal” numerals and “external” numerals (strings) denoting the same value in $\mathcal{R}$ and converted to the same target datatype should be converted to the same value. Indeed, the requirement on such conversions to round to nearest implies this. But even if this requirements is relaxed by a binding (see Annex A), external and internal conversions should not differ.

Negative values (including negative 0, $\mathbf{-0}$, if avaliable) can be obtained by using the negation operation (neg_F).

Radices other than 10 may also be available for floating point numerals.

Integer numerals may also be floating point numerals, i.e. their syntaxes need not be different. Nor need syntaxes for numerals for different floating point datatypes be different, nor need they be the same. This part does not specify the syntax for numerals. That is an issue for bindings or programming language specifications.

Overflow or underflow for floating point numerals can be detected at “compile time”, and warned about. Likewise can notifications about **infinitary** or **invalid**, e.g. for infinitary or NaN numerals that cannot be converted to the target type, be detected at “compile time” and be warned about.

B.6 Notification

An intermediate overflow on computing approximations to, for example, x^2 or y^2 during the calculation of $hypot_F(x, y) \approx \sqrt{x^2 + y^2}$ does not result in an overflow notification, unless the end result overflows. This is clear from the specification of the $hypot_F$ operation in this part. It is not helpful for the user of an operation to let intermediary overflows or underflows that are not reflected in the end result be propagated. Implementations of LIA-2 operations are required to shield the user from such intermediary overflows for all of the LIA-2 operations. More generally, well-made numerical software should similarly shield users of that software from overflows and underflows that are not reflected in a properly calculated end result. However, such requirements in general are beyond the scope of LIA-2.

If a single argument operation op_F, for the corresponding mathematical function f, is such that $f(x)$ very closely approximates x, when $|x| \leqslant fminN_F$, then $op_F(x)$ returns x for $|x| \leqslant fminN_F$, and does not give a notification if there cannot be any denormalisation loss relative to $f(x)$. For details, see the individual operation specifications for $expm1_F$, $ln1p_F$, sin_F, $arcsin_F$, tan_F, $arctan_F$, $sinh_F$, $arcsinh_F$, $tanh_F$, and $arctanh_F$.

Operations specified in LIA-2 return **invalid**(**qNaN**) when passed a signalling **NaN** (**sNaN**) as an argument. Most operations specified in LIA-2 return **qNaN**, without any notification when passed a quiet **NaN** (**qNaN**) as an argument.

The different kinds of notifications occur under the following circumstances:

a) **invalid**: when an argument is not valid for the operation, and no value in F^* or any special value result makes mathematical sense.

b) **infinitary**: when the input operand corresponds to a pole of the mathematical function approximated by the operation, or, more generally, when the true result is infinitary, but none of the arguments is infinitary.

c) **overflow**: when the (rounded) result is outside of the range of the result datatype.

d) **underflow**: when a sufficiently closely approximating result of the operation has a magnitude that is so small that it might not be sufficiently accurately represented in the result datatype.

e) **absolute_precision_underflow**: when the magnitude of the angle argument to a floating point trigonometric operation exceeds the maximum value of the argument for which the density of floating point values is deemed sufficient for the operation to make sense. See clause 5.3.7 and the associated discussion in this rationale (section B.5.3.7).

   In order to avoid **absolute_precision_underflow** notifications, and to maintain a high accuracy, implementors are encouraged to provide, and programmers encouraged to use, the angle normalisation operations specified in 5.3.8.1 and 5.3.9.1.

The difference between the **infinitary** and **overflow** notifications for floating point operations is that the first corresponds to a true mathematical singularity, and the second corresponds to a well-defined mathematical result that happens to lie outside the range of F.

B.6.1 Continuation values

For handling of notifications, the method that does recording of indicators (LIA-1, clause 6.1.2) is preferred.

An implementation which supports recording of indicators must supply continuation values to be used when execution is continued following the occurrence of a notification. For systems which support signed zeros, infinities and **NaN**s, LIA-2 specifies how these values, as well as ordinary values, are used as continuation values. Other implementations which use recording of indicators must supply other suitable continuation values and document the values selected.

B.7 Relationship with language standards

The datatypes involved in implicit conversions need not be accessible to the programmer. For example, trigonometric operations may be evaluated in extended double precision, even though that datatype is not made available to programmers using a particular programming language. These extra datatypes should be made available, however, and the implicit conversions should be expressible as explicit conversions. At least in order to be able to show exactly which expression is going to be evaluated without having to look at the machine code.

B.8 Documentation requirements

To make good use of an implementation that conforms to LIA-2, programmers need to know not only that the implementation conforms, but *how* it conforms. LIA-2 requires implementations to document the binding between the LIA-2 operations and parameters and the total arithmetic environment provided by the implementation.

It is expected that an implementation will meet part of its documentation requirements by incorporation of the relevant language standard. However, there will be aspects of the implementation that the language standard does not specify in the required detail, and the implementation needs to document those details. For example, the language standard may specify the range of allowed parameter values, but the implementation must document the actual value. The combination of the language standard and the implementation documentation together should meet all the requirements in clause 8.

Annex C
(informative)

Example bindings for specific languages

This annex describes how a computing system can simultaneously conform to a language standard (or publicly available specification) and to LIA-2. It contains suggestions for binding the "abstract" operations specified in LIA-2 to concrete language syntax. The format used for these example bindings in this annex is a short form version, suitable for the purposes of this annex. An actual binding is under no obligation to follow this format. An actual binding should, however, as in the bindings examples, give the LIA-2 operation name, or parameter name, bound to an identifier (or expression) by the binding.

Portability of programs can be improved if two conforming LIA-2 systems using the same programming language agree in the manner with which they adhere to LIA-2. For instance, LIA-2 requires that the parameter $big\_angle\_r_F$ be provided (if any conforming radian trigonometric operations are provided), but if one system provides it by means of the identifier `BigAngle` and another by the identifier `MaxAngle`, portability is impaired. Clearly, it would be best if such names were defined in the relevant language standards or binding standards, but in the meantime, suggestions are given here to aid portability. Name consistency cannot, however, be fully maintained between different programming languages, due to already existing differences in naming conventions, and LIA does *not* require wholesale naming changes, nor expression syntax changes.

The following clauses are suggestions rather than requirements because the areas covered are the responsibility of the various programming language standards committees. Until binding standards are in place, implementors can promote "de facto" portability by following these suggestions on their own.

The languages covered in this annex are

Ada,
BASIC,
C,
C++,
Fortran,
Haskell,
Java,
Common Lisp,
ISLisp,
Modula-2,
Pascal and Extended Pascal,
PL/I, and
SML.

This list is not exhaustive. Other languages and other computing devices (like 'scientific' calculators, 'web script' languages, and database 'query languages') are suitable for conformity to LIA-2.

In this annex, the parameters, operations, and exception behaviour of each language are examined to see how closely they fit the requirements of LIA-2. Where parameters, constants, or operations are not provided by the language, names and syntax are suggested. (Already provided syntax is marked with a ⋆.)

This annex describes only the language-level support for LIA-2. An implementation that wishes to conform must ensure that the underlying hardware and software is also configured to conform to LIA-2 requirements.

A complete binding for LIA-2 will include, or refer to, a binding for LIA-1. In turn, a complete binding for the LIA-1 may include, or refer to, a binding for IEC 60559. Such a joint LIA-2/LIA-1/IEC 60559 binding should be developed as a single binding standard. To avoid conflict with ongoing development, only the LIA-2 specific portions of such a binding are examplified in this annex.

Most language standards permit an implementation to provide, by some means, the parameters and operations required by LIA-2 that are not already part of the language. The method for accessing these additional parameters and operations depends on the implementation and language, and is not specified in LIA-2 nor examplified in this annex. It could include external subroutine libraries; new intrinsic functions supported by the compiler; constants and functions provided as global "macros"; and so on. The actual method of access through libraries, macros, etc. should of course be given in a real binding.

Most language standards do not constrain the accuracy of elementary numerical functions, or specify the subsequent behaviour after an arithmetic notification occurs.

In the event that there is a conflict between the requirements of the language standard and the requirements of LIA-2, the language binding standard should clearly identify the conflict and state its resolution of the conflict.

C.1 Ada

The programming language Ada is defined by ISO/IEC 8652:1995, *Information Technology – Programming Languages – Ada* [11].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to LIA-2 for that operation or parameter. For each of the marked items a suggested identifier is provided.

The Ada datatype `Boolean` corresponds to the LIA datatype **Boolean**.

Every implementation of Ada has at least one integer datatype, and at least one floating point datatype. The notations *INT* and *FLT* are used to stand for the names of one of these datatypes (respectively) in what follows.

Ada has an overloading system, so that the same name can be used for different types of arguments to the operations. Ada allows in general that formal parameter names are used in calls, though one normally does not write out the formal parameter names in calls. However, in some cases a formal parameter name is needed in the call to make the overloaded name resolve to the appropriate definition, rather than some other definition.

The Ada packages which contain some of the operations listed below are not detailed in this abbreviated example binding. For such details, see ISO/IEC 8652:1995, *Information Technology –*

Programming Languages – Ada [11]. A full binding would include information regarding packages also for the operations that are not included in the Ada standard.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	*INT*'`Max(`x`, `y`)`	⋆
$min_I(x, y)$	*INT*'`Min(`x`, `y`)`	⋆
$max\_seq_I(xs)$	`Max(`xs`)`	†
$min\_seq_I(xs)$	`Min(`xs`)`	†
$dim_I(x, y)$	`Dim(`x`, `y`)`	†
$power_I(x, y)$	x `**` y	⋆
$shift2_I(x, y)$	`Shift2(`x`, `y`)`	†
$shift10_I(x, y)$	`Shift10(`x`, `y`)`	†
$sqrt_I(x)$	`Sqrt(`x`)`	†
$divides_I(x, y)$	x `/= 0 and then` y `mod` x `= 0`	⋆
$even_I(x)$	x `mod 2 = 0`	⋆
$odd_I(x)$	x `mod 2 /= 0`	⋆
$quot_I(x, y)$	`Quotient(`x`, `y`)`	†
$mod_I(x, y)$	x `mod` y	⋆
$ratio_I(x, y)$	`Ratio(`x`, `y`)`	†
$residue_I(x, y)$	`Residue(`x`, `y`)`	†
$group_I(x, y)$	`Group(`x`, `y`)`	†
$pad_I(x, y)$	`Pad(`x`, `y`)`	†
$gcd_I(x, y)$	`Gcd(`x`, `y`)`	†
$lcm_I(x, y)$	`Lcm(`x`, `y`)`	†
$gcd\_seq_I(xs)$	`Gcd(`xs`)`	†
$lcm\_seq_I(xs)$	`Lcm(`xs`)`	†
$add\_wrap_I(x, y)$	`Add_Wrap(`x`, `y`)`	†
$add\_ov_I(x, y)$	`Add_Over(`x`, `y`)`	†
$sub\_wrap_I(x, y)$	`Sub_Wrap(`x`, `y`)`	†
$sub\_ov_I(x, y)$	`Sub_Over(`x`, `y`)`	†
$mul\_wrap_I(x, y)$	`Mul_Wrap(`x`, `y`)`	†
$mul\_ov_I(x, y)$	`Mul_Over(`x`, `y`)`	†

where x and y are expressions of type *INT* and where xs is an expression of type `array (Integer range <>) of` *INT*.

The LIA-2 basic floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	*FLT*'`Max(`x`, `y`)`	⋆
$min_F(x, y)$	*FLT*'`Min(`x`, `y`)`	⋆
$mmax_F(x, y)$	`MMax(`x`, `y`)`	†
$mmin_F(x, y)$	`MMin(`x`, `y`)`	†
$max\_seq_F(xs)$	`Max(`xs`)`	†
$min\_seq_F(xs)$	`Min(`xs`)`	†

$mmax\_seq_F(xs)$	`MMax(`xs`)`	†
$mmin\_seq_F(xs)$	`MMin(`xs`)`	†
$dim_F(x, y)$	`Dim(`x`, `y`)`	†
$floor_F(x)$	*FLT*`'Floor(`x`)`	⋆
$floor\_rest_F(x)$	x `-` *FLT*`'Floor(`x`)`	⋆
$rounding_F(x)$	*FLT*`'Unbiased_Rounding(`x`)`	⋆
$rounding\_rest_F(x)$	x `-` *FLT*`'Unbiased_Rounding(`x`)`	⋆
$ceiling_F(x)$	*FLT*`'Ceiling(`x`)`	⋆
$ceiling\_rest_F(x)$	x `-` *FLT*`'Ceiling(`x`)`	⋆
$residue_F(x, y)$	*FLT*`'Remainder(`x`, `y`)`	⋆
$sqrt_F(x)$	`Sqrt(`x`)`	⋆
$rec\_sqrt_F(x)$	`Rec_Sqrt(`x`)`	†
$mul_{F \to F'}(x, y)$	`Prod(`x`, `y`)`	†
$add\_lo_F(x, y)$	`Add_Low(`x`, `y`)`	†
$sub\_lo_F(x, y)$	`Sub_Low(`x`, `y`)`	†
$mul\_lo_F(x, y)$	`Mul_Low(`x`, `y`)`	†
$div\_rest_F(x, y)$	`Div_Rest(`x`, `y`)`	†
$sqrt\_rest_F(x)$	`Sqrt_Rest(`x`)`	†

where x and y are expressions of type *FLT*, and where xs is an expression of type `array (Integer range <>) of` *FLT*.

The parameters for LIA-2 operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`Err_Hypotenuse(`x`)`	†
$max\_error\_exp_F$	`Err_Exp(`x`)`	†
$max\_error\_power_F$	`Err_Power(`x`)`	†
$big\_angle\_r_F$	`Big_Radian_Angle(`x`)`	†
$max\_error\_rad_F$	`Err_Rad(`x`)`	†
$max\_error\_sin_F$	`Err_Sin(`x`)`	†
$max\_error\_tan_F$	`Err_Tan(`x`)`	†
$min\_angular\_unit_F$	`Smallest_Angular_Unit(`x`)`	†
$big\_angle\_u_F$	`Big_Angle(`x`)`	†
$max\_error\_sinu_F(u)$	`Err_Sin_Cycle(`u`)`	†
$max\_error\_tanu_F(u)$	`Err_Tan_Cycle(`u`)`	†
$max\_error\_sinh_F$	`Err_Sinh(`x`)`	†
$max\_error\_tanh_F$	`Err_Tanh(`x`)`	†
$max\_error\_convert_F$	`Err_Convert(`x`)`	†
$max\_error\_convert_{F'}$	`Err_Convert_To_String`	†
$max\_error\_convert_{D'}$	`Err_Convert_To_String`	†

where x and u are expressions of type *FLT*, and F' and D' are non-special value sets for string formats. Several of the parameter functions are constant for each type (and library), the argument is then used only to differentiate among the floating point types. (This is in Ada normally done as 'type attributes', but new such cannot be defined outside of the Ada standard itself.)

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`Hypotenuse(`x`, `y`)`	†
$power_{F,I}(b, z)$	b `**` z	⋆
$exp_F(x)$	`Exp(`x`)`	⋆
$expm1_F(x)$	`ExpM1(`x`)`	†
$exp2_F(x)$	`Exp2(`x`)`	†
$exp10_F(x)$	`Exp10(`x`)`	†
$power_F(b, y)$	b `**` y	⋆
$power1pm1_F(b, y)$	`Power1PM1(`b`, `y`)`	†
$ln_F(x)$	`Log(`x`)`	⋆
$ln1p_F(x)$	`Log1P(`x`)`	†
$log2_F(x)$	`Log2(`x`)`	†
$log10_F(x)$	`Log10(`x`)`	†
$logbase_F(b, x)$	`Log(`x`, `b`)` (note parameter order)	⋆
$logbase1p1p_F(b, x)$	`Log1P1P(`x`, `b`)`	†
$rad_F(x)$	`Rad(`x`)`	†
$axis\_rad_F(x)$	`Rad(`x`, `h`, `v`)` (note out parameters)	†
$sin_F(x)$	`Sin(`x`)`	⋆
$cos_F(x)$	`Cos(`x`)`	⋆
$tan_F(x)$	`Tan(`x`)`	⋆
$cot_F(x)$	`Cot(`x`)`	⋆
$sec_F(x)$	`Sec(`x`)`	†
$csc_F(x)$	`Csc(`x`)`	†
$cossin_F(x)$	`CosSin(`x`, `c`, `s`)` (note out parameters)	†
$arcsin_F(x)$	`ArcSin(`x`)`	⋆
$arccos_F(x)$	`ArcCos(`x`)`	⋆
$arctan_F(x)$	`ArcTan(`x`)`	⋆
$arccot_F(x)$	`ArcCotS(`x`)`	†
$arccotc_F(x)$	`ArcCot(`x`)`	⋆
$arcsec_F(x)$	`ArcSec(`x`)`	†
$arccsc_F(x)$	`ArcCsc(`x`)`	†
$arc_F(x, y)$	`ArcTan(`y`, `x`)` or `ArcCot(`x`, `y`)`	⋆(**invalid** at origin)
$cycle_F(u, x)$	`Cycle(`x`, `u`)` (note parameter order)	†
$axis\_cycle_F(u, x)$	`Cycle(`x`, `u`, `h`, `v`)`	†
$sinu_F(u, x)$	`Sin(`x`, `u`)` (note parameter order)	⋆
$cosu_F(u, x)$	`Cos(`x`, `u`)`	⋆
$tanu_F(u, x)$	`Tan(`x`, `u`)`	⋆

$cotu_F(u, x)$	`Cot(`x`, `u`)`	⋆
$secu_F(u, x)$	`Sec(`x`, `u`)`	†
$cscu_F(u, x)$	`Csc(`x`, `u`)`	†
$cossinu_F(u, x)$	`CosSin(`x`, `u`, `c`, `s`)`	†
$arcsinu_F(u, x)$	`ArcSin(`x`, `u`)`	⋆
$arccosu_F(u, x)$	`ArcCos(`x`, `u`)`	⋆
$arctanu_F(u, x)$	`ArcTan(`x`, Cycle=>`u`)`	⋆
$arccotu_F(u, x)$	`ArcCotS(`x`, `u`)`	†
$arccotcu_F(u, x)$	`ArcCot(`x`, Cycle=>`u`)`	⋆
$arcsecu_F(u, x)$	`ArcSec(`x`, `u`)`	†
$arccscu_F(u, x)$	`ArcCsc(`x`, `u`)`	†
$arcu_F(u, x, y)$	`ArcTan(`y`, `x`, `u`)` or `ArcCot(`x`, `y`, `u`)`	⋆(**invalid** at origin)
$rad\_to\_cycle_F(x, w)$	`Rad_to_Cycle(`x`, `w`)`	†
$cycle\_to\_rad_F(u, x)$	`Cycle_to_Rad(`u`, `x`)`	†
$cycle\_to\_cycle_F(u, x, w)$	`Cycle_to_Cycle(`u`, `x`, `w`)`	†
$sinh_F(x)$	`SinH(`x`)`	⋆
$cosh_F(x)$	`CosH(`x`)`	⋆
$tanh_F(x)$	`TanH(`x`)`	⋆
$coth_F(x)$	`CotH(`x`)`	⋆
$sech_F(x)$	`SecH(`x`)`	†
$csch_F(x)$	`CscH(`x`)`	†
$arcsinh_F(x)$	`ArcSinH(`x`)`	⋆
$arccosh_F(x)$	`ArcCosH(`x`)`	⋆
$arctanh_F(x)$	`ArcTanH(`x`)`	⋆
$arccoth_F(x)$	`ArcCotH(`x`)`	⋆
$arcsech_F(x)$	`ArcSecH(`x`)`	†
$arccsch_F(x)$	`ArcCscH(`x`)`	†

where b, x, y, u, and w are expressions of type *FLT*, z is an expression of type *INT*, and c, s, h, and v are variables of type *FLT*.

Ada95 specifies (in other words) that $power_{F,I}$ must be computed by repeated multiplication (in an unspecified order). That computation method cannot, whatever the order of multiplications, guarantee fulfillment of the LIA-2 accuracy requirements, and cannot fulfill the required relationship between $power_{F,I}$ and $power_F$. Further, Ada95 specifies that angular units must be positive, and implicitly has a value for $min\_angular\_unit_F$ of $fmin_F$. LIA-2 allows also negative angular units, but has a larger value for $min\_angular\_unit_F$. A real Ada binding for LIA-2 must state how these conflicts are resolved (see clause 2).

Arithmetic value conversions in Ada are always explicit and usually use the destination datatype name as the name of the conversion function, except when converting to/from string formats.

$convert_{I \to I'}(x)$	*INT2*`(`x`)`	⋆
$convert_{I'' \to I}(s)$	`Get(`s`, `n`, `w`);`	⋆
$convert_{I'' \to I}(f)$	`Get(`f`?, `n`, `w`?);`	⋆
$convert_{I \to I''}(x)$	`Put(`s`, `x`, `*base*`?);`	⋆
$convert_{I \to I''}(x)$	`Put(`h`?, `x`, `w`?, `*base*`?);`	⋆

$floor_{F\rightarrow I}(y)$	*INT*(*FLT*'Floor(*y*))	⋆
$rounding_{F\rightarrow I}(y)$	*INT*(*FLT*'Unbiased_Rounding(*y*))	⋆
$ceiling_{F\rightarrow I}(y)$	*INT*(*FLT*'Ceiling(*y*))	⋆
$convert_{I\rightarrow F}(x)$	*FLT*(*x*)	⋆
$convert_{F\rightarrow F'}(y)$	*FLT2*(*y*)	⋆
$convert_{F''\rightarrow F}(s)$	Get(*s*, *n*, *w*?);	⋆
$convert_{F''\rightarrow F}(f)$	Get(*f*?, *n*, *w*?);	⋆
$convert_{F\rightarrow F''}(y)$	Put(*s*, *y*, Aft=>*a*?, Exp=>*e*?);	⋆
$convert_{F\rightarrow F''}(y)$	Put(*h*?, *y*, Fore=>*i*?, Aft=>*a*?, Exp=>*e*?);	⋆
$convert_{D\rightarrow F}(z)$	*FLT*(*z*)	⋆
$convert_{D'\rightarrow F}(s)$	Get(*s*, *n*, *w*?);	⋆
$convert_{D'\rightarrow F}(f)$	Get(*f*?, *n*, *w*?);	⋆
$convert_{F\rightarrow D}(y)$	*FXD*(*y*)	⋆
$convert_{F\rightarrow D'}(y)$	Put(*s*, *y*, Aft=>*a*?, Exp=>0);	⋆
$convert_{F\rightarrow D'}(y)$	Put(*h*?, *y*, Fore=>*i*?, Aft=>*a*?, Exp=>0);	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*, and z is an expression of type *FXD*, where *FXD* is a fixed point type. *INT2* is the integer datatype that corresponds to I'. *FLT2* is the floating point datatype that corresponds to F'. A ? above indicates that the parameter is optional. f is an opened input file (default is the default input file). h is an opened output file (default is the default output file). s is of type `String` or `Wide_String`. For `Get` of a floating point or fixed point numeral, the base is indicated in the numeral (default 10). For `Put` of a floating point or fixed point numeral, only base 10 is required to be supported. For details on `Get` and `Put`, see clause A.10.8 Input-Output for Integer Types, A.10.9 Input-Output for Real Types, and A.11 Wide Text Input-Output, of ISO/IEC 8652:1995. *base*, n, w, i, a, and e are expressions for non-negative integers. e is greater than 0. *base* is greater than 1.

Ada provides non-negative numerals for all its integer and floating point types. The default base is 10, but all bases from 2 to 16 can be used. There is no differentiation between the numerals for different floating point types, nor between numerals for different integer types, but integer numerals (without a point) cannot be used for floating point types, and 'real' numerals (with a point) cannot be used for integer types. Integer numerals can have an exponent part though. The details are not repeated in this example binding, see ISO/IEC 8652:1995, clause 2.4 Numeric Literals, clause 3.5.4 Integer Types, and clause 3.5.6 Real Types.

The Ada standard does not specify any numerals for infinities and NaNs. Suggestion:

$+\infty$	*FLT*'Infinity	†
qNaN	*FLT*'NaN	†
sNaN	*FLT*'NaNSignalling	†

as well as string formats for reading and writing these values as character strings.

Ada has a notion of 'exception' that implies a non-returnable, but catchable, change of control flow. Ada uses its exception mechanism as its default means of notification. **underflow** does not cause any notification in Ada, and the continuation value to the **underflow** is used directly, since an Ada exception is inappropriate for an **underflow** notification. On **underflow** the continuation value (specified in LIA-2) is used directly without recording the **underflow** itself. Ada uses the exception `Constraint_Error` for **infinitary** and **overflow** notifications, and the exceptions `Numerics.Argument_Error`, `IO_Exceptions.Data_Error`, and `IO_Exceptions.End_Error` for **invalid** notifications. Since Ada exceptions are non-returnable changes of control flow, no continuation value is provided for these notifications.

An implementation that wishes to follow LIA-2 should provide recording of indicators as an alternative means of handling numeric notifications. Recording of indicators is the LIA-2 preferred means of handling numeric notifications.

C.2 BASIC

The programming language BASIC is defined by ANSI X3.113-1987 (R1998) [40], endorsed by ISO/IEC 10279:1991, *Information technology – Programming languages – Full BASIC* [16].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

BASIC has no user accessible datatype corresponding to the LIA datatype **Boolean**.

BASIC has one primitive computational datatype, `numeric`. The model presented by the BASIC language is that of a real number with decimal radix and a specified (minimum) number of significant decimal digits. Numeric data is not declared directly, but any special characteristics are inferred from how they are used and from any `OPTIONS` that are in force.

The BASIC statement `OPTION ARITHMETIC NATIVE` ties the `numeric` type more closely to the underlying implementation. The precision and type of `NATIVE` numeric data is implementation dependent.

For the trigonometric operations, if `OPTION ANGLE DEGREES` is in effect, the argument or result is given in degrees. If `OPTION ANGLE RADIANS` (default) is in effect, the argument or result is given in radians.

Since the BASIC numeric datatype does not match LIA-1 integer datatypes, this binding example does not include any of the LIA-2 operations for integer datatypes.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	`MAX(`x`, `y`)`	⋆
$min_F(x, y)$	`MIN(`x`, `y`)`	⋆
$mmax_F(x, y)$	`MMAX(`x`, `y`)`	†
$mmin_F(x, y)$	`MMIN(`x`, `y`)`	†
$max\_seq_F(xs)$	`MAXS(`xs`)`	†
$min\_seq_F(xs)$	`MINS(`xs`)`	†
$mmax\_seq_F(xs)$	`MMAXS(`xs`)`	†
$mmin\_seq_F(xs)$	`MMINS(`xs`)`	†
$dim_F(x, y)$	`MONUS(`x`, `y`)`	†
$floor_F(x)$	`INT(`x`)`	⋆

$floor\_rest_F(x)$	`x - INT(x)`	⋆
$rounding_F(x)$	`ROUND(x)`	⋆
$rounding\_rest_F(x)$	`x - ROUND(x)`	⋆
$ceiling_F(x)$	`CEIL(x)`	⋆
$ceiling\_rest_F(x)$	`x - CEIL(x)`	⋆
$residue_F(x, y)$	`RESIDUE(x, y)`	†
$sqrt_F(x)$	`SQR(x)`	⋆
$rec\_sqrt_F(x)$	`REC_SQRT(x)`	†
$add\_lo_F(x, y)$	`ADD_LOW(x, y)`	†
$sub\_lo_F(x, y)$	`SUB_LOW(x, y)`	†
$mul\_lo_F(x, y)$	`MUL_LOW(x, y)`	†
$div\_rest_F(x, y)$	`DIV_REST(x, y)`	†
$sqrt\_rest_F(x)$	`SQRT_REST(x)`	†

where x and y are expressions of type `numeric`, and where xs is an expression of type array of `numeric`.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`ERR_HYPOTENUSE`	†
$max\_error\_exp_F$	`ERR_EXP`	†
$max\_error\_power_F$	`ERR_POWER`	†
$big\_angle\_r_F$	`BIG_RADIAN_ANGLE`	†
$max\_error\_rad_F$	`ERR_RAD`	†
$max\_error\_sin_F$	`ERR_SIN`	†
$max\_error\_tan_F$	`ERR_TAN`	†
$min\_angular\_unit_F$	`MIN_ANGLE_UNIT`	†
$big\_angle\_u_F$	`BIG_ANGLE`	†
$max\_error\_sinu_F(u)$	`ERR_SIN_CYCLE(u)`	†
$max\_error\_tanu_F(u)$	`ERR_TAN_CYCLE(u)`	†
$max\_error\_sinh_F$	`ERR_SINH`	†
$max\_error\_tanh_F$	`ERR_TANH`	†
$max\_error\_convert_F$	`ERR_CONVERT`	†
$max\_error\_convert_{F'}$	`ERR_CONVERT_TO_STRING`	†
$max\_error\_convert_{D'}$	`ERR_CONVERT_TO_STRING`	†

where u is an expression of type `numeric`.

The LIA-2 floating point operations are listed below, along with the syntax used to invoke them. BASIC has a degree mode and a radian mode for the trigonometric operations.

$hypot_F(x, y)$	`HYPOT(x, y)`	†
$exp_F(x)$	`EXP(x)`	⋆
$expm1_F(x)$	`EXPM1(x)`	†

$exp2_F(x)$	`EXP2(`x`)`	†
$exp10_F(x)$	`EXP10(`x`)`	†
$power_F(b, y)$	`POWER(`b`, `y`)`	†
$power1pm1_F(b, y)$	`POWER1PM1(`b`, `y`)`	†
$ln_F(x)$	`LOG(`x`)`	⋆
$ln1p_F(x)$	`LOG1P(`x`)`	†
$log2_F(x)$	`LOG2(`x`)`	⋆
$log10_F(x)$	`LOG10(`x`)`	⋆
$logbase_F(b, x)$	`LOGBASE(`b`, `x`)`	†
$logbase1p1p_F(b, x)$	`LOGBASE1P1P(`b`, `x`)`	†
$rad_F(x)$	`NORMANGLE(`x`)` (when in radian mode)	†
$sin_F(x)$	`SIN(`x`)` (when in radian mode)	⋆
$cos_F(x)$	`COS(`x`)` (when in radian mode)	⋆
$tan_F(x)$	`TAN(`x`)` (when in radian mode)	⋆
$cot_F(x)$	`COT(`x`)` (when in radian mode)	⋆
$sec_F(x)$	`SEC(`x`)` (when in radian mode)	⋆
$csc_F(x)$	`CSC(`x`)` (when in radian mode)	⋆
$arcsin_F(x)$	`ASIN(`x`)` (when in radian mode)	⋆
$arccos_F(x)$	`ACOS(`x`)` (when in radian mode)	⋆
$arctan_F(x)$	`ATN(`x`)` (when in radian mode)	⋆
$arccot_F(x)$	`ACOT(`x`)` (when in radian mode)	†
$arccotc_F(x)$	`ACOTC(`x`)` (when in radian mode)	†
$arcsec_F(x)$	`ASEC(`x`)` (when in radian mode)	†
$arccsc_F(x)$	`ACSC(`x`)` (when in radian mode)	†
$arc_F(x, y)$	`ANGLE(`x`, `y`)` (when in radian mode)	⋆(**invalid** at origin)
$cycle_F(u, x)$	`NORMANGLEU(`u`, `x`)`	†
$sinu_F(u, x)$	`SINU(`u`, `x`)`	†
$cosu_F(u, x)$	`COSU(`u`, `x`)`	†
$tanu_F(u, x)$	`TANU(`u`, `x`)`	†
$cotu_F(u, x)$	`COTU(`u`, `x`)`	†
$secu_F(u, x)$	`SECU(`u`, `x`)`	†
$cscu_F(u, x)$	`CSCU(`u`, `x`)`	†
$arcsinu_F(u, x)$	`ASINU(`u`, `x`)`	†
$arccosu_F(u, x)$	`ACOSU(`u`, `x`)`	†
$arctanu_F(u, x)$	`ATNU(`u`, `x`)`	†
$arccotu_F(u, x)$	`ACOTU(`u`, `x`)`	†
$arccotcu_F(u, x)$	`ACOTCU(`u`, `x`)`	†
$arcsecu_F(u, x)$	`ASECU(`u`, `x`)`	†
$arccscu_F(u, x)$	`ACSCU(`u`, `x`)`	†
$arcu_F(u, x, y)$	`ANGLEU(`u`, `x`, `y`)`	†

$cycle_F(360, x)$	`NORMANGLE(x)` (when in degree mode)	†
$sinu_F(360, x)$	`SIN(x)` (when in degree mode)	$\star$
$cosu_F(360, x)$	`COS(x)` (when in degree mode)	$\star$
$tanu_F(360, x)$	`TAN(x)` (when in degree mode)	$\star$
$cotu_F(360, x)$	`COT(x)` (when in degree mode)	$\star$
$secu_F(360, x)$	`SEC(x)` (when in degree mode)	$\star$
$cscu_F(360, x)$	`CSC(x)` (when in degree mode)	$\star$
$arcsinu_F(360, x)$	`ASIN(x)` (when in degree mode)	$\star$
$arccosu_F(360, x)$	`ACOS(x)` (when in degree mode)	$\star$
$arctanu_F(360, x)$	`ATN(x)` (when in degree mode)	$\star$
$arccotu_F(360, x)$	`ACOT(x)` (when in degree mode)	†
$arccotcu_F(360, x)$	`ACOTC(x)` (when in degree mode)	†
$arcsecu_F(360, x)$	`ASEC(x)` (when in degree mode)	†
$arccscu_F(360, x)$	`ACSC(x)` (when in degree mode)	†
$arcu_F(360, x, y)$	`ANGLE(x, y)` (when in degree mode)	$\star$(**invalid** at origin)
$rad\_to\_cycle_F(x, 360)$	`DEG(x)`	$\star$
$cycle\_to\_rad_F(360, x)$	`RAD(x)`	$\star$
$rad\_to\_cycle_F(x, w)$	`RAD_TO_CYCLE(x, w)`	†
$cycle\_to\_rad_F(u, x)$	`CYCLE_TO_RAD(u, x)`	†
$cycle\_to\_cycle_F(u, x, w)$	`CYCLE_TO_CYCLE(u, x, w)`	†
$sinh_F(x)$	`SINH(x)`	$\star$
$cosh_F(x)$	`COSH(x)`	$\star$
$tanh_F(x)$	`TANH(x)`	$\star$
$coth_F(x)$	`COTH(x)`	†
$sech_F(x)$	`SECH(x)`	†
$csch_F(x)$	`CSCH(x)`	†
$arcsinh_F(x)$	`ARCSINH(x)`	†
$arccosh_F(x)$	`ARCCOSH(x)`	†
$arctanh_F(x)$	`ARCTANH(x)`	†
$arccoth_F(x)$	`ARCCOTH(x)`	†
$arcsech_F(x)$	`ARCSECH(x)`	†
$arccsch_F(x)$	`ARCCSCH(x)`	†

where b, x, y, u, and w are expressions of type `numeric`.

Arithmetic value conversions in BASIC are always tied to reading and writing text.

$convert_{F''\to F}(stdin)$	`READ x`	$\star$
$convert_{F\to F''}(y)$	`PRINT y`	$\star$
$convert_{D'\to F}(stdin)$	`READ x`	$\star$

where x is a variable of type `numeric`, and y is an expression of type `numeric`.

BASIC provides non-negative numerals for `numeric` in base 10.

BASIC does not specify any numerals for infinities and NaNs. Suggestion:

$+\infty$	`INFINITY`	†
qNaN	`NAN`	†
sNaN	`NANSIGNALLING`	†

as well as string formats for reading and writing these values as character strings.

BASIC has a notion of 'exception' that implies a non-returnable change of control flow. BASIC uses its exception mechanism as its default means of notification. **underflow** does not cause any notification in BASIC, and the continuation value to the **underflow** is used directly, since an BASIC exception is inappropriate for an **underflow** notification. BASIC uses the exception numbers `1001` to `1008` for **overflow**, exception numbers `1502` and `1503` for *handled* **underflow**, the exception number `3001` to `3004` for **infinitary**, the exception numbers `-3000`, `3002`, and `3004` to `3011` for **invalid**, and the exception numbers `-3050` and `-3051` for **absolute_precision_underflow**. Since BASIC exceptions are non-returnable changes of control flow, no continuation value is provided for these notifications, except that *unhandled* underflow uses the continuation value specified without any BASIC exception.

An implementation that wishes to follow LIA-2 should provide recording of indicators as an alternative means of handling numeric notifications. Recording of indicators is the LIA-2 preferred means of handling numeric notifications.

C.3 C

The programming language C is defined by ISO/IEC 9899:1999, *Information technology – Programming languages – C* [17]. This edition of the C standard is often referred to as C99, which is also used below.

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

The LIA datatype **Boolean** is implemented by the C datatype `int` (1 = **true** and 0 = **false**) or the new C99 `_Bool` datatype.

C defines numerous integer datatypes. They may be aliases of each other in an implementation defined way. The description here is not complete. See the C99 standard. Some of the integer datatypes have a predetermined bit width, and the signed ones use 2's complement for representation of negative values: `int`*n*`_t` and `uint`*n*`_t`, where *n* is the bit width expressed as a decimal numeral. Some bit widths are required. There are also minimum width, fastest minimum width, and special purpose integer datatypes (like `size_t`). Also provided are the more well-known integer datatypes `char`, `short int`, `int`, `long int`, `long long int` (new in C99), `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int` (new in C99). Finally there are the integer datatypes `intmax_t` and `uintmax_t` (both new in C99) that are the largest provided signed and unsigned integer datatypes. `intmax_t` and `uintmax_t` may even be unbounded with a negative integer infinity as `INTMAX_MIN` and a positive integer infinity as `INTMAX_MAX` and `UINTMAX_MAX`. *INT* is used below to designate one of the integer datatypes.

C names three floating point datatypes: `float`, `double`, and `long double`. *FLT* is used below to designate one of the floating point datatypes.

For some of the operations below, the C standard defines 'type generic macros'. These are fixed by the C standard, and new ones cannot be defined in program libraries.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	`imaxt(x, y)`	†
$max_I(x, y)$	`(x < y ? y : x)`	⋆
$min_I(x, y)$	`imint(x, y)`	†
$min_I(x, y)$	`(x < y ? x : y)`	⋆
$max\_seq_I(xs)$	`imax_arrt(xs, nr_of_items)`	†
$min\_seq_I(xs)$	`imin_arrt(xs, nr_of_items)`	†
$dim_I(x, y)$	`idimt(x, y)`	†
$power_I(x, y)$	`ipowert(x, y)`	†
$shift2_I(x, y)$	`shift2t(x, y)`	†
$shift10_I(x, y)$	`shift10t(x, y)`	†
$sqrt_I(x)$	`isqrtt(x)`	†
$divides_I(x, y)$	`does_dividet(x, y)`	†
$divides_I(x, y)$	`x != 0 && y % x == 0`	⋆
$even_I(x)$	`x % 2 == 0`	⋆
$odd_I(x)$	`x % 2 != 0`	⋆
$quot_I(x, y)$	`quott(x, y)`	†
$mod_I(x, y)$	`modt(x, y)`	†
$ratio_I(x, y)$	`ratiot(x, y)`	†
$residue_I(x, y)$	`iremaindert(x, y)`	†
$group_I(x, y)$	`groupt(x, y)`	†
$pad_I(x, y)$	`padt(x, y)`	†
$gcd_I(x, y)$	`gcdt(x, y)`	†
$lcm_I(x, y)$	`lcmt(x, y)`	†
$gcd\_seq_I(xs)$	`gcd_arrt(xs, nr_of_items)`	†
$lcm\_seq_I(xs)$	`lcm_arrt(xs, nr_of_items)`	†
$add\_wrap_I(x, y)$	`add_wrapt(x, y)`	†
$add\_ov_I(x, y)$	`add_overt(x, y)`	†
$sub\_wrap_I(x, y)$	`sub_wrapt(x, y)`	†
$sub\_ov_I(x, y)$	`sub_overt(x, y)`	†
$mul\_wrap_I(x, y)$	`mul_wrapt(x, y)`	†
$mul\_ov_I(x, y)$	`mul_overt(x, y)`	†

where x and y are expressions of the same integer type and where xs is an expression of type array of an integer type. t is a string (part of the operation name in C), that is the empty string for `int`, is `l` for `long int`, is `u` for `unsigned int`, and is `ul` for `unsigned long int`.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	`nmaxt(x, y)`	†
$min_F(x, y)$	`nmint(x, y)`	†

$mmax_F(x, y)$	`fmax`t`(`x`, `y`)` or `fmax(`x`, `y`)`	⋆(C99)
$mmin_F(x, y)$	`fmin`t`(`x`, `y`)` or `fmin(`x`, `y`)`	⋆(C99)
$max\_seq_F(xs)$	`nmax_arr`t`(`xs`, `nr_of_items`)`	†
$min\_seq_F(xs)$	`nmin_arr`t`(`xs`, `nr_of_items`)`	†
$mmax\_seq_F(xs)$	`fmax_arr`t`(`xs`, `nr_of_items`)`	†
$mmin\_seq_F(xs)$	`fmin_arr`t`(`xs`, `nr_of_items`)`	†
$dim_F(x, y)$	`fdim`t`(`x`, `y`)` or `fdim(`x`, `y`)` (dev. for special values)	⋆(C99)
$floor_F(x)$	`floor`t`(`x`)` or `floor(`x`)`	⋆
$floor\_rest_F(x)$	x `- floor`t`(`x`)`	⋆
$rounding_F(x)$	`nearbyint`t`(`x`)` (when in round to nearest mode)	⋆(C99)
$rounding\_rest_F(x)$	x `- nearbyint`t`(`x`)` (when in round to nearest mode)	⋆(C99)
$ceiling_F(x)$	`ceil`t`(`x`)` or `ceil(`x`)`	⋆
$ceiling\_rest_F(x)$	x `- ceil`t`(`x`)`	⋆
$residue_F(x, y)$	`remainder`t`(`x`, `y`)` or `remainder(`x`, `y`)`	⋆(C99)
$sqrt_F(x)$	`sqrt`t`(`x`)` or `sqrt(`x`)`	⋆
$rec\_sqrt_F(x)$	`rec_sqrt`t`(`x`)`	†
$mul_{F \to F'}(x, y)$	`dprod`t`(`x`, `y`)`	†
$add\_lo_F(x, y)$	`add_low`t`(`x`, `y`)`	†
$sub\_lo_F(x, y)$	`sub_low`t`(`x`, `y`)`	†
$mul\_lo_F(x, y)$	`mul_low`t`(`x`, `y`)`	†
$div\_rest_F(x, y)$	`div_rest`t`(`x`, `y`)`	†
$sqrt\_rest_F(x)$	`sqrt_rest`t`(`x`)`	†

where x and y are expressions of the same floating point type, and where xs is an expression of type array of a floating point type. t is a string (part of the operation name), that is the empty string for `double`, is `f` for `float`, and is `l` for `long double` (the same applies to the parameters and operations below).

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`err_hypot`t	†
$max\_error\_exp_F$	`err_exp`t	†
$max\_error\_power_F$	`err_power`t	†
$big\_angle\_r_F$	`big_radian_angle`t	†
$max\_error\_rad_F$	`err_rad`t	†
$max\_error\_sin_F$	`err_sin`t	†
$max\_error\_tan_F$	`err_tan`t	†
$min\_angular\_unit_F$	`smallest_angle_unit`t	†
$big\_angle\_u_F$	`big_angle`t	†
$max\_error\_sinu_F(u)$	`err_sin_cycle`t`(`u`)`	†
$max\_error\_tanu_F(u)$	`err_tan_cycle`t`(`u`)`	†
$max\_error\_sinh_F$	`err_sinh`t	†
$max\_error\_tanh_F$	`err_tanh`t	†

where u is an expression of a floating point type. No conversion error parameters are needed, since C99 requires all floating point datatype conversion (even to and from strings) to always have an error that is $\leqslant 0.5$ ulp.

C has a `pow` operation that does not conform to LIA-2, but may be specified in LIA-2 terms:

$$
\begin{array}{lll}
pow_F(x,y) & = power_{F\mathcal{Z}}(x,y) & \text{if } y \in F \cap \mathcal{Z} \\
 & = pow_F(x,0) & \text{if } y = \mathbf{-0} \\
 & = 1 & \text{if } x \in \{-1,1\} \text{ and } y \in \{\mathbf{-\infty},\mathbf{+\infty}\} \\
 & = 1 & \text{if } x \text{ is a quiet NaN and } y = 0 \\
 & = power_F(x,y) & \text{otherwise}
\end{array}
$$

C99 has a `hypot` operation that does not conform to LIA-2, but may be specified in LIA-2 terms:

$$
\begin{array}{lll}
hhypot_F(x,y) & = \mathbf{+\infty} & \text{if } x \text{ is a quiet NaN and } y \in \{\mathbf{-\infty},\mathbf{+\infty}\} \\
 & = \mathbf{+\infty} & \text{if } x \in \{\mathbf{-\infty},\mathbf{+\infty}\} \text{ and } y \text{ is a quiet NaN} \\
 & = hypot_F(x,y) & \text{otherwise}
\end{array}
$$

The LIA-2 elementary floating point operations are listed below, together with the non-LIA-2 pow_F and $hhypot_F$, along with the syntax used to invoke them:

$hypot_F(x,y)$	`hypotenuse`t`(`x`, `y`)`	†
$hhypot_F(x,y)$	`hypot`t`(`x`, `y`)` or `hypot(`x`, `y`)`	⋆ Not LIA-2!
$power_{F,I}(b,z)$	`poweri`t`(`b`, `z`)`	†
$exp_F(x)$	`exp`t`(`x`)` or `exp(`x`)`	⋆
$expm1_F(x)$	`expm1`t`(`x`)` or `expm1(`x`)`	⋆(C99)
$exp2_F(x)$	`exp2`t`(`x`)` or `exp2(`x`)`	⋆(C99)
$exp10_F(x)$	`exp10`t`(`x`)`	†
$power_F(b,y)$	`power`t`(`b`, `y`)`	†
$pow_F(b,y)$	`pow`t`(`b`, `y`)` or `pow(`b`, `y`)`	⋆ Not LIA-2!
$power1pm1_F(b,y)$	`power1pm1`t`(`b`, `y`)`	†
$ln_F(x)$	`log`t`(`x`)` or `log(`x`)`	⋆
$ln1p_F(x)$	`log1p`t`(`x`)` or `log1p(`x`)`	⋆(C99)
$log2_F(x)$	`log2`t`(`x`)` or `log2(`x`)`	⋆(C99)
$log10_F(x)$	`log10`t`(`x`)` or `log10(`x`)`	⋆
$logbase_F(b,x)$	`logbase`t`(`b`, `x`)`	†
$logbase1p1p_F(b,x)$	`logbase1p1p`t`(`b`, `x`)`	†
$rad_F(x)$	`radian`t`(`x`)`	†
$axis\_rad_F(x)$	`axis_rad`t`(`x`, &`h`, &`v`)` (note out parameters)	†
$sin_F(x)$	`sin`t`(`x`)` or `sin(`x`)`	⋆
$cos_F(x)$	`cos`t`(`x`)` or `cos(`x`)`	⋆
$tan_F(x)$	`tan`t`(`x`)` or `tan(`x`)`	⋆
$cot_F(x)$	`cot`t`(`x`)`	†
$sec_F(x)$	`sec`t`(`x`)`	†
$csc_F(x)$	`csc`t`(`x`)`	†
$cossin_F(x)$	`cossin`t`(`x`, &`c`, &`s`)`	†

$arcsin_F(x)$	`asint(x)` or `asin(x)`	⋆
$arccos_F(x)$	`acost(x)` or `acos(x)`	⋆
$arctan_F(x)$	`atant(x)` or `atan(x)`	⋆
$arccot_F(x)$	`acott(x)`	†
$arccotc_F(x)$	`acotct(x)`	†
$arcsec_F(x)$	`asect(x)`	†
$arccsc_F(x)$	`acsct(x)`	†
$arc_F(x, y)$	`atan2t(y, x)` or `atan2(y, x)`	⋆
$cycle_F(u, x)$	`cyclet(u, x)`	†
$axis\_cycle_F(u, x)$	`axis_cyclet(u, x, &h, &v)`	†
$sinu_F(u, x)$	`sinut(u, x)`	†
$cosu_F(u, x)$	`cosut(u, x)`	†
$tanu_F(u, x)$	`tanut(u, x)`	†
$cotu_F(u, x)$	`cotut(u, x)`	†
$secu_F(u, x)$	`secut(u, x)`	†
$cscu_F(u, x)$	`cscut(u, x)`	†
$cossinu_F(u, x)$	`cossinut(u, x, &c, &s)`	†
$arcsinu_F(u, x)$	`asinut(u, x)`	†
$arccosu_F(u, x)$	`acosut(u, x)`	†
$arctanu_F(u, x)$	`atanut(u, x)`	†
$arccotu_F(u, x)$	`acotut(u, x)`	†
$arccctcu_F(u, x)$	`acotcut(u, x)`	†
$arcsecu_F(u, x)$	`asecut(u, x)`	†
$arccscu_F(u, x)$	`acscut(u, x)`	†
$arcu_F(u, x, y)$	`atan2ut(u, y, x)`	†
$rad\_to\_cycle_F(x, w)$	`radian_to_cyclet(x, w)`	†
$cycle\_to\_rad_F(u, x)$	`cycle_to_radiant(u, x)`	†
$cycle\_to\_cycle_F(u, x, w)$	`cycle_to_cyclet(u, x, w)`	†
$sinh_F(x)$	`sinht(x)` or `sinh(x)`	⋆(C99)
$cosh_F(x)$	`cosht(x)` or `cosh(x)`	⋆(C99)
$tanh_F(x)$	`tanht(x)` or `tanh(x)`	⋆(C99)
$coth_F(x)$	`cotht(x)`	†
$sech_F(x)$	`secht(x)`	†
$csch_F(x)$	`cscht(x)`	†
$arcsinh_F(x)$	`asinht(x)` or `asinh(x)`	⋆(C99)
$arccosh_F(x)$	`acosht(x)` or `acosh(x)`	⋆(C99)
$arctanh_F(x)$	`atanht(x)` or `atanh(x)`	⋆(C99)
$arccoth_F(x)$	`acotht(x)`	†
$arcsech_F(x)$	`asecht(x)`	†
$arccsch_F(x)$	`acscht(x)`	†

where b, x, y, u, and w are expressions of type *FLT*, h, v, c, and s are lvalue expressions of type *FLT*, and z is an expression of type *INT*.

Arithmetic value conversions in C can be explicit or implicit. The explicit arithmetic value conversions are usually expressed as 'casts', except when converting to/from string formats. The rules for when implicit conversions are applied is not repeated here, but work as if a cast had been applied.

When converting to/from string formats, format strings are used. The format string is used as a pattern for the string format generated or parsed. The description of format strings here is not complete. Please see the C99 standard for a full description. In the format strings `%` is used to indicate the start of a format pattern. After the `%`, optionally a string field width (w below) may be given as a positive decimal integer numeral. For the floating and fixed point format patterns, there may then optionally be a '`.`' followed by a positive integer numeral (d below) indicating the number of fractional digits in the string. The C operations below use HYPHEN-MINUS rather than MINUS (which would have been typographically better), and only digits that are in ASCII, independently of so-called locale. For generating or parsing other kinds of digits, say Arabic digits or Thai digits, another API must be used, that is not standardised in C. For the floating and fixed point formats, $+\infty$ may be represented as either `inf` or `infinity`, $-\infty$ may be represented as either `-inf` or `-infinity`, and a **NaN** may be represented as `NaN`; all independently of so-called locale. For language dependent representations of these values another API must be used, that is not standardised in C.

For the integer formats then follows an internal type indicator, of which some are new to C99. Not all C99 integer types have internal type indicators. However, for t below, `hh` indicates `char`, `h` indicates `short int`, the empty string indicates `int`, `l` (the letter l) indicates `long int`, `ll` (the letters ll) indicates `long long int`, and `j` indicates `intmax_t` or `uintmax_t`. (For system purposes there are also special type names like `size_t`, and `z` indicates `size_t` and `t` indicates `ptrdiff_t` as type format letters.) Finally, there is a radix (for the string side) and signedness (both sides) format letter (r below): `d` for signed decimal; `o`, `u`, `x`, `X` for octal, decimal, hexadecimal with small letters, and hexadecimal with capital letters, all unsigned. E.g., `%jd` indicates decimal numeral string for `intmax_t`, `%2hhx` indicates hexadecimal numeral string for `unsigned char`, with a two character field width, and `%lu` indicates decimal numeral string for `unsigned long int`.

For the floating point formats instead follows another internal type indicator. Not all C99 floating point types have standard internal type indicators for the format strings. However, for u below the empty string indicates `double` and `L` indicates `long double`. Finally, there is a radix (for the string side) format letter: `e` or `E` for decimal, `a` or `A` for hexadecimal. E.g., `%15.8LA` indicates hexadecimal floating point numeral string for `long double`, with capital letters for the letter components, a field width of 15 characters, and 8 hexadecimal fractional digits.

For the fixed point formats also follows the internal type indicator as for the floating point formats. But for the final part of the pattern, there is another radix (for the string side) format letter (p below), only two are standardised, both for the decimal radix: `f` or `F`. E.g., `%Lf` indicates decimal fixed point numeral string for `long double`, with a small letter for the letter component. (There is also a combined floating/fixed point string format: `g`.)

$convert_{I \to I'}(x)$	$(INT2)\,x$	⋆
$convert_{I'' \to I}(s)$	`sscanf(`s`, "%`wtr`", &`i`)`	⋆
$convert_{I'' \to I}(f)$	`fscanf(`f`, "%`wtr`", &`i`)`	⋆
$convert_{I \to I''}(x)$	`sprintf(`s`, "%`wtr`", `x`)`	⋆
$convert_{I \to I''}(x)$	`fprintf(`h`, "%`wtr`", `x`)`	⋆
$floor_{F \to I}(y)$	(INT)`floor`t`(`y`)`	⋆

$floor_{F\to I}(y)$	(*INT*)`nearbyint`*t*(*y*)	(when in round towards $-\infty$ mode)	⋆(C99)
$rounding_{F\to I}(y)$	(*INT*)`nearbyint`*t*(*y*)	(when in round to nearest mode)	⋆(C99)
$ceiling_{F\to I}(y)$	(*INT*)`nearbyint`*t*(*y*)	(when in round towards $+\infty$ mode)	⋆(C99)
$ceiling_{F\to I}(y)$	(*INT*)`ceil`*t*(*y*)		⋆
$convert_{I\to F}(x)$	(*FLT*)*x*		⋆
$convert_{F\to F'}(y)$	(*FLT2*)*y*		⋆
$convert_{F''\to F}(s)$	`sscanf(`*s*`, "%`*w*`.`*duv*`", &`*r*`)`		⋆
$convert_{F''\to F}(f)$	`fscanf(`*f*`, "%`*w*`.`*duv*`", &`*r*`)`		⋆
$convert_{F\to F''}(y)$	`sprintf(`*s*`, "%`*w*`.`*duv*`", `*y*`)`		⋆
$convert_{F\to F''}(y)$	`fprintf(`*h*`, "%`*w*`.`*duv*`", `*y*`)`		⋆
$convert_{D'\to F}(s)$	`sscanf(`*s*`, "%`*wup*`", &`*g*`)`		⋆
$convert_{D'\to F}(f)$	`fscanf(`*f*`, "%`*wup*`", &`*g*`)`		⋆
$convert_{F\to D'}(y)$	`sprintf(`*s*`, "%`*w*`.`*dup*`", `*y*`)`		⋆
$convert_{F\to D'}(y)$	`fprintf(`*h*`, "%`*w*`.`*dup*`", `*y*`)`		⋆

where s is an expression of type `char*`, f is an expression of type `FILE*`, i is an lvalue expression of type `int`, g is an lvalue expression of type `double`, x is an expression of type *INT*, y is an expression of type *FLT*, *INT2* is the integer datatype that corresponds to I', and *FLT2* is the floating point datatype that corresponds to F'.

C99 provides non-negative numerals for all its integer and floating point types. The default base is 10, but base 8 (for integers) and 16 (both integer and float) can be used too. Numerals for different integer types are distinguished by suffixes. Numerals for different floating point types are distinguished by suffix: `f` for `float`, no suffix for `double`, `l` for `long double`. Numerals for floating point types must have a '.' or an exponent in them. The details are not repeated in this example binding, see ISO/IEC 9899:1999, clause 6.4.4.1 Integer constants, and clause 6.4.4.2 Floating constants.

C99 specifies numerals (as macros) for infinities and NaNs for `float`:

$+\infty$	`INFINITY`	⋆
qNaN	`NAN`	⋆
sNaN	`NANSIGNALLING`	†

as well as string formats for reading and writing these values as character strings.

C99 has two ways of handling arithmetic errors. One, for backwards compatibility, is by assigning to `errno`. The other is by recording of indicators, the method preferred by LIA-2, which can be used for floating point errors. For C99, the **absolute_precision_underflow** notification is ignored. The behaviour when integer operations initiate a notification is, however, not defined by C99.

C.4 C++

The programming language C++ is defined by ISO/IEC 14882:1998, *Programming languages – C++* [18].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

This example binding recommends that all identifiers suggested here be defined in the namespace `std::math`.

The LIA datatype **Boolean** is implemented by the C++ datatype `bool`.

Every implementation of C++ has integral datatypes `int`, `long int`, `unsigned int`, and `unsigned long int`. *INT* is used below to designate one of the integer datatypes.

C++ has three floating point datatypes: `float`, `double`, and `long double`. *FLT* is used below to designate one of the floating point datatypes.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x,y)$	`max(`x`, `y`)`	⋆
$min_I(x,y)$	`min(`x`, `y`)`	⋆
$max\_seq_I(xs)$	xs`.max()`	⋆
$min\_seq_I(xs)$	xs`.min()`	⋆
$dim_I(x,y)$	`dim(`x`, `y`)`	†
$power_I(x,y)$	`power(`x`, `y`)`	†
$shift2_I(x,y)$	`shift2(`x`, `y`)`	†
$shift10_I(x,y)$	`shift10(`x`, `y`)`	†
$sqrt_I(x)$	`sqrt(`x`)`	†
$divides_I(x,y)$	`does_divide(`x`, `y`)`	†
$divides_I(x,y)$	x `!= 0 &&` y `%` x `== 0`	⋆
$even_I(x)$	x `% 2 == 0`	⋆
$odd_I(x)$	x `% 2 != 0`	⋆
$quot_I(x,y)$	`quot(`x`, `y`)`	†
$mod_I(x,y)$	`mod(`x`, `y`)`	†
$ratio_I(x,y)$	`ratio(`x`, `y`)`	†
$residue_I(x,y)$	`remainder(`x`, `y`)`	†
$group_I(x,y)$	`group(`x`, `y`)`	†
$pad_I(x,y)$	`pad(`x`, `y`)`	†
$gcd_I(x,y)$	`gcd(`x`, `y`)`	†
$lcm_I(x,y)$	`lcm(`x`, `y`)`	†
$gcd\_seq_I(xs)$	xs`.gcd()`	†
$lcm\_seq_I(xs)$	xs`.lcm()`	†
$add\_wrap_I(x,y)$	`add_wrap(`x`, `y`)`	†
$add\_ov_I(x,y)$	`add_over(`x`, `y`)`	†
$sub\_wrap_I(x,y)$	`sub_wrap(`x`, `y`)`	†
$sub\_ov_I(x,y)$	`sub_over(`x`, `y`)`	†
$mul\_wrap_I(x,y)$	`mul_wrap(`x`, `y`)`	†

$mul\_ov_I(x,y)$	`mul_over(`x`, `y`)`	†

where x and y are expressions of the same integer type and where xs is an expression of type valarray of an integer type.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x,y)$	`nmax(`x`, `y`)`	†
$min_F(x,y)$	`nmin(`x`, `y`)`	†
$mmax_F(x,y)$	`max(`x`, `y`)`	⋆
$mmin_F(x,y)$	`min(`x`, `y`)`	⋆
$max\_seq_F(xs)$	xs`.nmax()`	†
$min\_seq_F(xs)$	xs`.nmin()`	†
$mmax\_seq_F(xs)$	xs`.max()`	⋆
$mmin\_seq_F(xs)$	xs`.min()`	⋆
$dim_F(x,y)$	`dim(`x`, `y`)`	†
$floor_F(x)$	`floor(`x`)`	⋆
$floor\_rest_F(x)$	x `- floor(`x`)`	⋆
$rounding_F(x)$	`round(`x`)`	†
$rounding\_rest_F(x)$	x `- round(`x`)`	†
$ceiling_F(x)$	`ceil(`x`)`	⋆
$ceiling\_rest_F(x)$	x `- ceil(`x`)`	⋆
$residue_F(x,y)$	`remainder(`x`, `y`)`	†
$sqrt_F(x)$	`sqrt(`x`)`	⋆
$rec\_sqrt_F(x)$	`rec_sqrt(`x`)`	†
$mul_{F \to F'}(x,y)$	`dprod(`x`, `y`)`	†
$add\_lo_F(x,y)$	`add_low(`x`, `y`)`	†
$sub\_lo_F(x,y)$	`sub_low(`x`, `y`)`	†
$mul\_lo_F(x,y)$	`mul_low(`x`, `y`)`	†
$div\_rest_F(x,y)$	`div_rest(`x`, `y`)`	†
$sqrt\_rest_F(x)$	`sqrt_rest(`x`)`	†

where x and y are expressions of the same floating point type, and where xs is an expression of type valarray of a floating point type. The C++ standard does not make clear how to handle NaN arguments, in particular for max and min.

The parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`numeric_limits<`*FLT*`>::err_hypotenuse()`	†
$max\_error\_exp_F$	`numeric_limits<`*FLT*`>::err_exp()`	†
$max\_error\_power_F$	`numeric_limits<`*FLT*`>::err_power()`	†
$big\_angle\_r_F$	`numeric_limits<`*FLT*`>::big_radian_angle()`	†
$max\_error\_rad_F$	`numeric_limits<`*FLT*`>::err_rad()`	†
$max\_error\_sin_F$	`numeric_limits<`*FLT*`>::err_sin()`	†
$max\_error\_tan_F$	`numeric_limits<`*FLT*`>::err_tan()`	†

$min\_angular\_unit_F$	`numeric_limits<`*FLT*`>::smallest_angle_unit()`	†
$big\_angle\_u_F$	`numeric_limits<`*FLT*`>::big_angle()`	†
$max\_error\_sinu_F(u)$	`numeric_limits<`*FLT*`>::err_sin_cycle(`*u*`)`	†
$max\_error\_tanu_F(u)$	`numeric_limits<`*FLT*`>::err_tan_cycle(`*u*`)`	†
$max\_error\_sinh_F$	`numeric_limits<`*FLT*`>::err_sinh()`	†
$max\_error\_tanh_F$	`numeric_limits<`*FLT*`>::err_tanh()`	†
$max\_error\_convert_F$	`numeric_limits<`*FLT*`>::err_convert()`	†
$max\_error\_convert_{F'}$	`numeric_limits<`*FLT*`>::err_convert_to_string()`	†
$max\_error\_convert_{D'}$	`numeric_limits<`*FLT*`>::err_convert_to_string()`	†

where u is an expression of a floating point type.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`hypotenuse(`*x*`, `*y*`)`	†
$power_{F,I}(b, z)$	`power(`*b*`, `*z*`)`	†
$exp_F(x)$	`exp(`*x*`)`	⋆
$expm1_F(x)$	`expm1(`*x*`)`	†
$exp2_F(x)$	`exp2(`*x*`)`	†
$exp10_F(x)$	`exp10(`*x*`)`	†
$power_F(b, y)$	`power(`*b*`, `*y*`)`	†
$pow_F(b, y)$	`pow(`*b*`, `*y*`)`	⋆ Not LIA-2! (See C.)
$power1pm1_F(b, y)$	`power1pm1(`*b*`, `*y*`)`	†
$ln_F(x)$	`log(`*x*`)`	⋆
$ln1p_F(x)$	`log1p(`*x*`)`	†
$log2_F(x)$	`log2(`*x*`)`	†
$log10_F(x)$	`log10(`*x*`)`	⋆
$logbase_F(b, x)$	`logbase(`*b*`, `*x*`)`	†
$logbase1p1p_F(b, x)$	`logbase1p1p(`*b*`, `*x*`)`	†
$rad_F(x)$	`rad(`*x*`)`	†
$axis\_rad_F(x)$	`axis_rad(`*x*`, &`*h*`, &`*v*`)` (note out parameters)	†
$sin_F(x)$	`sin(`*x*`)`	⋆
$cos_F(x)$	`cos(`*x*`)`	⋆
$tan_F(x)$	`tan(`*x*`)`	⋆
$cot_F(x)$	`cot(`*x*`)`	†
$sec_F(x)$	`sec(`*x*`)`	†
$csc_F(x)$	`csc(`*x*`)`	†
$cossin_F(x)$	`cossin(`*x*`, &`*c*`, &`*s*`)`	†
$arcsin_F(x)$	`asin(`*x*`)`	⋆
$arccos_F(x)$	`acos(`*x*`)`	⋆
$arctan_F(x)$	`atan(`*x*`)`	⋆
$arccot_F(x)$	`acot(`*x*`)`	†

$arccotc_F(x)$	`acotc(`x`)`	†
$arcsec_F(x)$	`asec(`x`)`	†
$arccsc_F(x)$	`acsc(`x`)`	†
$arc_F(x, y)$	`atan2(`y`, `x`)`	⋆
$cycle_F(u, x)$	`cycle(`u`, `x`)`	†
$axis\_cycle_F(u, x)$	`axis_cycle(`u`, `x`, &`h`, &`v`)`	†
$sinu_F(u, x)$	`sinu(`u`, `x`)`	†
$cosu_F(u, x)$	`cosu(`u`, `x`)`	†
$tanu_F(u, x)$	`tanu(`u`, `x`)`	†
$cotu_F(u, x)$	`cotu(`u`, `x`)`	†
$secu_F(u, x)$	`secu(`u`, `x`)`	†
$cscu_F(u, x)$	`cscu(`u`, `x`)`	†
$cossinu_F(x)$	`cossinu(`u`, `x`, &`c`, &`s`)`	†
$arcsinu_F(u, x)$	`asinu(`u`, `x`)`	†
$arccosu_F(u, x)$	`acosu(`u`, `x`)`	†
$arctanu_F(u, x)$	`atanu(`u`, `x`)`	†
$arccotu_F(u, x)$	`acotu(`u`, `x`)`	†
$arccotcu_F(u, x)$	`acotcu(`u`, `x`)`	†
$arcsecu_F(u, x)$	`asecu(`u`, `x`)`	†
$arccscu_F(u, x)$	`acscu(`u`, `x`)`	†
$arcu_F(u, x, y)$	`atan2u(`u`, `y`, `x`)`	†
$rad\_to\_cycle_F(x, w)$	`radian_to_cycle(`x`, `w`)`	†
$cycle\_to\_rad_F(u, x)$	`cycle_to_radian(`u`, `x`)`	†
$cycle\_to\_cycle_F(u, x, w)$	`cycle_to_cycle(`u`, `x`, `w`)`	†
$sinh_F(x)$	`sinh(`x`)`	⋆
$cosh_F(x)$	`cosh(`x`)`	⋆
$tanh_F(x)$	`tanh(`x`)`	⋆
$coth_F(x)$	`coth(`x`)`	†
$sech_F(x)$	`sech(`x`)`	†
$csch_F(x)$	`csch(`x`)`	†
$arcsinh_F(x)$	`asinh(`x`)`	†
$arccosh_F(x)$	`acosh(`x`)`	†
$arctanh_F(x)$	`atanh(`x`)`	†
$arccoth_F(x)$	`acoth(`x`)`	†
$arcsech_F(x)$	`asech(`x`)`	†
$arccsch_F(x)$	`acsch(`x`)`	†

where b, x, y, u, and w are expressions of type *FLT*, h, v, c, and s are lvalue expressions of type *FLT*, and z is an expression of type *INT*.

Arithmetic value conversions in C++ can be explicit or implicit. The rules for when implicit conversions are applied are not repeated here. C++ also deals with stream input/output in other ways, see clause 22.2.2 of ISO/IEC 14882:1998, 'Locale and facets'. The explicit arithmetic value conversions are usually expressed as 'casts', except when converting to/from string formats.

When converting to/from string formats, format strings are used. The format string is used as a pattern for the string format generated or parsed. The description of format strings here is not complete. Please see the C++ standard for a full description. In the format strings `%` is used to indicate the start of a format pattern. After the `%`, optionally a string field width (w below) may be given as a positive decimal integer numeral. For the floating and fixed point format patterns, there may then optionally be a '`.`' followed by a positive integer numeral (d below) indicating the number of fractional digits in the string. The C++ operations below use HYPHEN-MINUS rather than MINUS (which would have been typographically better), and only digits that are in ASCII, independently of so-called locale. For generating or parsing other kinds of digits, say Arabic digits or Thai digits, another API must be used, that is not standardised in C++. For the floating and fixed point formats, $+\infty$ may be represented as either `inf` or `infinity`, $-\infty$ may be represented as either `-inf` or `-infinity`, and a **NaN** may be represented as `NaN`; all independently of so-called locale. For language dependent representations of these values another API must be used, that is not standardised in C.

For the integer formats then follows an internal type indicator. For t below, the empty string indicates `int`, `l` (the letter l) indicates `long int`. Finally, there is a radix (for the string side) and signedness (both sides) format letter (r below): `d` for signed decimal; `o`, `u`, `x`, `X` for octal, decimal, hexadecimal with small letters, and hexadecimal with capital letters, all unsigned. E.g., `%d` indicates decimal numeral string for `int` and `%lu` indicates decimal numeral string for `unsigned long int`.

For the floating point formats instead follows another internal type indicator. For u below the empty string indicates `double` and `L` indicates `long double`. Finally, there is a radix (for the string side) format letter: `e` or `E` for decimal. E.g., `%15.8LE` indicates hexadecimal floating point numeral string for `long double`, with a capital letter for the letter component, a field width of 15 characters, and 8 hexadecimal fractional digits.

For the fixed point formats also follows the internal type indicator as for the floating point formats. But for the final part of the pattern, there is another radix (for the string side) format letter (p below), only two are standardised, both for the decimal radix: `f` or `F`. E.g., `%Lf` indicates decimal fixed point numeral string for `long double`, with a small letter for the letter component. (There is also a combined floating/fixed point string format: `g`.)

$convert_{I \to I'}(x)$	`static_cast<`*INT2*`>(`x`)`	⋆
$convert_{I'' \to I}(s)$	`sscanf(`s`, "%`wtr`", &`i`)`	⋆
$convert_{I'' \to I}(f)$	`fscanf(`f`, "%`wtr`", &`i`)`	⋆
$convert_{I \to I''}(x)$	`sprintf(`s`, "%`wtr`", `x`)`	⋆
$convert_{I \to I''}(x)$	`fprintf(`h`, "%`wtr`", `x`)`	⋆
$floor_{F \to I}(y)$	`static_cast<`*INT*`>(floor(`y`))`	⋆
$rounding_{F \to I}(y)$	`static_cast<`*INT*`>(round(`y`))`	†
$ceiling_{F \to I}(y)$	`static_cast<`*INT*`>(ceil(`y`))`	⋆
$convert_{I \to F}(x)$	`static_cast<`*FLT*`>(`x`)`	⋆
$convert_{F \to F'}(y)$	`(`*FLT2*`)`y	⋆
$convert_{F'' \to F}(s)$	`sscanf(`s`, "%`$w.duv$`", &`r`)`	⋆
$convert_{F'' \to F}(f)$	`fscanf(`f`, "%`$w.duv$`", &`r`)`	⋆

$convert_{F \to F''}(y)$	`sprintf(`s`, "%`w`.`duv`", `y`)`	⋆
$convert_{F \to F''}(y)$	`fprintf(`h`, "%`w`.`duv`", `y`)`	⋆
$convert_{D' \to F}(s)$	`sscanf(`s`, "%`wup`", &`g`)`	⋆
$convert_{D' \to F}(f)$	`fscanf(`f`, "%`wup`", &`g`)`	⋆
$convert_{F \to D'}(y)$	`sprintf(`s`, "%`w`.`dup`", `y`)`	⋆
$convert_{F \to D'}(y)$	`fprintf(`h`, "%`w`.`dup`", `y`)`	⋆

where s is an expression of type `char*`, f is an expression of type `FILE*`, i is an lvalue expression of type `int`, g is an lvalue expression of type `double`, x is an expression of type *INT*, y is an expression of type *FLT*, *INT2* is the integer datatype that corresponds to I', and *FLT2* is the floating point datatype that corresponds to F'.

C++ provides non-negative numerals for all its integer and floating point types in base 10. Numerals for different integer types are distinguished by suffixes. Numerals for different floating point types are distinguished by suffix: `f` for `float`, no suffix for `double`, `l` for `long double`. Numerals for floating point types must have a '.' or an exponent in them. The details are not repeated in this example binding, see ISO/IEC 14882:1998, clause 2.9.1 Integer literals, and clause 2.9.4 Floating literals.

C++ specifies numerals for infinities and NaNs:

$+\infty$	`numeric_limits<`*FLT*`>::infinity()`	⋆
qNaN	`numeric_limits<`*FLT*`>::quiet_NaN()`	⋆
sNaN	`numeric_limits<`*FLT*`>::signaling_NaN()`	⋆

as well as string formats for reading and writing these values as character strings.

C++ has completely undefined behaviour on arithmetic notification. An implementation wishing to conform to LIA-2 should provide a means for recording of indicators, similar to C99.

C.5 Fortran

The programming language Fortran is defined by ISO/IEC 1539-1:1997, *Information technology – Programming languages – Fortran – Part 1: Base language* [22]. It is complemented with ISO/IEC TR 15580:1998, *Information technology – Programming languages – Fortran – Floating-point exception handling* [23].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided. The operations marked "(⋆)" are not part of the base standard, but included in the *Floating-point exception handling* Technical Report [23].

The Fortran datatype `LOGICAL` corresponds to the LIA datatype **Boolean**.

Every implementation of Fortran has one integer datatype, denoted as `INTEGER`, and two floating point datatype denoted as `REAL` (single precision) and `DOUBLE PRECISION`.

An implementation is permitted to offer additional `INTEGER` types with a different range and additional `REAL` types with different precision or range, each parameterised with a *kind* parameter.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	`MAX(`x`, `y`)`	⋆
$min_I(x, y)$	`MIN(`x`, `y`)`	⋆
$max\_seq_I(xs)$	`MAX(`x_1`, `x_2`, ..., `x_n`)`	⋆
$max\_seq_I(xs)$	`MAXVAL(`xs`)`	⋆
$min\_seq_I(xs)$	`MIN(`x_1`, `x_2`, ..., `x_n`)`	⋆
$min\_seq_I(xs)$	`MINVAL(`xs`)`	⋆
$dim_I(x, y)$	`DIM(`x`, `y`)`	⋆
$power_I(x, y)$	x `**` y	⋆
$shift2_I(x, y)$	`SHIFT2(`x`, `y`)`	†
$shift10_I(x, y)$	`SHIFT10(`x`, `y`)`	†
$sqrt_I(x)$	`ISQRT(`x`)`	†
$divides_I(x, y)$	`DIVIDES(`x`, `y`)`	†
$even_I(x)$	`MODULO(`x`,2) == 0`	⋆
$odd_I(x)$	`MODULO(`x`,2) /= 0`	⋆
$quot_I(x, y)$	`QUOTIENT(`x`, `y`)`	†
$mod_I(x, y)$	`MODULO(`x`, `y`)`	⋆
$ratio_I(x, y)$	`RATIO(`x`, `y`)`	†
$residue_I(x, y)$	`RESIDUE(`x`, `y`)`	†
$group_I(x, y)$	`GROUP(`x`, `y`)`	†
$pad_I(x, y)$	`PAD(`x`, `y`)`	†
$gcd_I(x, y)$	`GCD(`x`, `y`)`	†
$lcm_I(x, y)$	`LCM(`x`, `y`)`	†
$gcd\_seq_I(xs)$	`GCDVAL(`xs`)`	†
$lcm\_seq_I(xs)$	`LCMVAL(`xs`)`	†
$add\_wrap_I(x, y)$	`ADD_WRAP(`x`, `y`)`	†
$add\_ov_I(x, y)$	`ADD_OVER(`x`, `y`)`	†
$sub\_wrap_I(x, y)$	`SUB_WRAP(`x`, `y`)`	†
$sub\_ov_I(x, y)$	`SUB_OVER(`x`, `y`)`	†
$mul\_wrap_I(x, y)$	`MUL_WRAP(`x`, `y`)`	†
$mul\_ov_I(x, y)$	`MUL_OVER(`x`, `y`)`	†

where x and y are expressions of type `INTEGER(`***`kind`***`)` and where xs is an expression of type array of `INTEGER(`***`kind`***`)`.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	`MAX(`x`, `y`)`	⋆
$min_F(x, y)$	`MIN(`x`, `y`)`	⋆
$mmax_F(x, y)$	`MMAX(`x`, `y`)`	†
$mmin_F(x, y)$	`MMIN(`x`, `y`)`	†
$max\_seq_F(xs)$	`MAX(`x_1`, `x_2`, ..., `x_n`)`	⋆
$max\_seq_F(xs)$	`MAXVAL(`xs`)`	⋆
$min\_seq_F(xs)$	`MIN(`x_1`, `x_2`, ..., `x_n`)`	⋆
$min\_seq_F(xs)$	`MINVAL(`xs`)`	⋆

$mmax\_seq_F(xs)$	`MMAX(`x_1`, `x_2`, ..., `x_n`)`	†
$mmax\_seq_F(xs)$	`MMAXVAL(`xs`)`	†
$mmin\_seq_F(xs)$	`MMIN(`x_1`, `x_2`, ..., `x_n`)`	†
$mmin\_seq_F(xs)$	`MMINVAL(`xs`)`	†
$dim_F(x, y)$	`DIM(`x`, `y`)`	⋆
$floor_F(x)$	`IEEE_RINT(`x`)` (if in round towards $-\infty$ mode)	(⋆)
$floor\_rest_F(x)$	x `- IEEE_RINT(`x`)` (if in round towards $-\infty$ mode)	(⋆)
$rounding_F(x)$	`IEEE_RINT(`x`)` (if in round to nearest mode)	(⋆)
$rounding\_rest_F(x)$	x `- IEEE_RINT(`x`)` (if in round to nearest mode)	(⋆)
$ceiling_F(x)$	`IEEE_RINT(`x`)` (if in round towards $+\infty$ mode)	(⋆)
$ceiling\_rest_F(x)$	x `- IEEE_RINT(`x`)` (if in round towards $+\infty$ mode)	(⋆)
$residue_F(x, y)$	`IEEE_REM(`x`, `y`)`	(⋆)
$sqrt_F(x)$	`SQRT(`x`)`	⋆
$rec\_sqrt_F(x)$	`REC_SQRT(`x`)`	†
$mul_{F \rightarrow F'}(x, y)$	`DPROD(`x`, `y`)`	⋆
$add\_lo_F(x, y)$	`ADD_LOW(`x`, `y`)`	†
$sub\_lo_F(x, y)$	`SUB_LOW(`x`, `y`)`	†
$mul\_lo_F(x, y)$	`MUL_LOW(`x`, `y`)`	†
$div\_rest_F(x, y)$	`DIV_REST(`x`, `y`)`	†
$sqrt\_rest_F(x)$	`SQRT_REST(`x`)`	†

where x and y are expressions of type `REAL(`***kind***`)`, and where xs is an expression of type array of `REAL(`***kind***`)`.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`ERR_HYPOTENUSE(`x`)`	†
$max\_error\_exp_F$	`ERR_EXP(`x`)`	†
$max\_error\_power_F$	`ERR_POWER(`x`)`	†
$big\_angle\_r_F$	`BIG_RADIAN_ANGLE(`x`)`	†
$max\_error\_rad_F$	`ERR_RAD(`x`)`	†
$max\_error\_sin_F$	`ERR_SIN(`x`)`	†
$max\_error\_tan_F$	`ERR_TAN(`x`)`	†
$min\_angular\_unit_F$	`MIN_ANGLE_UNIT(`x`)`	†
$big\_angle\_u_F$	`BIG_ANGLE(`x`)`	†
$max\_error\_sinu_F(u)$	`ERR_SIN_CYCLE(`u`)`	†
$max\_error\_tanu_F(u)$	`ERR_TAN_CYCLE(`u`)`	†
$max\_error\_sinh_F$	`ERR_SINH(`x`)`	†
$max\_error\_tanh_F$	`ERR_TANH(`x`)`	†
$max\_error\_convert_F$	`ERR_CONVERT(`x`)`	†
$max\_error\_convert_{F'}$	`ERR_CONVERT_TO_STRING`	†
$max\_error\_convert_{D'}$	`ERR_CONVERT_TO_STRING`	†

where x and u are expressions of type `REAL(kind)`. Several of the parameter functions are constant for each type (and library), the argument is then used only to differentiate among the floating point types.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`HYPOT(`x`, `y`)`	†
$power_{F,I}(b, z)$	b `**` z	⋆
$exp_F(x)$	`EXP(`x`)`	⋆
$expm1_F(x)$	`EXPM1(`x`)`	†
$exp2_F(x)$	`EXP2(`x`)`	†
$exp10_F(x)$	`EXP10(`x`)`	†
$power_F(b, y)$	b `**` y	⋆
$power1pm1_F(b, y)$	`POWER1PM1(`b`, `y`)`	†
$ln_F(x)$	`LOG(`x`)`	⋆
$ln1p_F(x)$	`LOG1P(`x`)`	†
$log2_F(x)$	`LOG2(`x`)`	†
$log10_F(x)$	`LOG10(`x`)`	⋆
$logbase_F(b, x)$	`LOGBASE(`b`, `x`)`	†
$logbase1p1p_F(b, x)$	`LOGBASE1P1P(`b`, `x`)`	†
$rad_F(x)$	`RAD(`x`)`	†
$sin_F(x)$	`SIN(`x`)`	⋆
$cos_F(x)$	`COS(`x`)`	⋆
$tan_F(x)$	`TAN(`x`)`	⋆
$cot_F(x)$	`COT(`x`)`	†
$sec_F(x)$	`SEC(`x`)`	†
$csc_F(x)$	`CSC(`x`)`	†
$arcsin_F(x)$	`ASIN(`x`)`	⋆
$arccos_F(x)$	`ACOS(`x`)`	⋆
$arctan_F(x)$	`ATAN(`x`)`	⋆
$arccot_F(x)$	`ACOT(`x`)`	†
$arccotc_F(x)$	`ACOTC(`x`)`	†
$arcsec_F(x)$	`ASEC(`x`)`	†
$arccsc_F(x)$	`ACSC(`x`)`	†
$arc_F(x, y)$	`ATAN2(`y`, `x`)`	⋆
$cycle_F(u, x)$	`CYCLE(`u`, `x`)`	†
$sinu_F(u, x)$	`SINU(`u`, `x`)`	†
$cosu_F(u, x)$	`COSU(`u`, `x`)`	†
$tanu_F(u, x)$	`TANU(`u`, `x`)`	†
$cotu_F(u, x)$	`COTU(`u`, `x`)`	†
$secu_F(u, x)$	`SECU(`u`, `x`)`	†

$cscu_F(u, x)$	`CSCU(`u`, `x`)`	†
$arcsinu_F(u, x)$	`ASINU(`u`, `x`)`	†
$arccosu_F(u, x)$	`ACOSU(`u`, `x`)`	†
$arctanu_F(u, x)$	`ATANU(`u`, `x`)`	†
$arccotu_F(u, x)$	`ACOTU(`u`, `x`)`	†
$arccotcu_F(u, x)$	`ACOTCU(`u`, `x`)`	†
$arcsecu_F(u, x)$	`ASECU(`u`, `x`)`	†
$arccscu_F(u, x)$	`ACSCU(`u`, `x`)`	†
$arcu_F(u, x, y)$	`ATAN2U(`u`, `y`, `x`)`	†
$cycle_F(360, x)$	`DEGREES(`x`)`	†
$sinu_F(360, x)$	`SIND(`x`)`	†
$cosu_F(360, x)$	`COSD(`x`)`	†
$tanu_F(360, x)$	`TAND(`x`)`	†
$cotu_F(360, x)$	`COTD(`x`)`	†
$secu_F(360, x)$	`SECD(`x`)`	†
$cscu_F(360, x)$	`CSCD(`x`)`	†
$arcsinu_F(360, x)$	`ASIND(`x`)`	†
$arccosu_F(360, x)$	`ACOSD(`x`)`	†
$arctanu_F(360, x)$	`ATAND(`x`)`	†
$arccotu_F(360, x)$	`ACOTD(`x`)`	†
$arccotcu_F(360, x)$	`ACOTCD(`x`)`	†
$arcsecu_F(360, x)$	`ASECD(`x`)`	†
$arccscu_F(360, x)$	`ACSCD(`x`)`	†
$arcu_F(360, x, y)$	`ATAN2D(`y`, `x`)`	†
$rad\_to\_cycle_F(x, w)$	`RAD_TO_CYCLE(`x`, `w`)`	†
$cycle\_to\_rad_F(u, x)$	`CYCLE_TO_RAD(`u`, `x`)`	†
$cycle\_to\_cycle_F(u, x, w)$	`CYCLE_TO_CYCLE(`u`, `x`, `w`)`	†
$sinh_F(x)$	`SINH(`x`)`	⋆
$cosh_F(x)$	`COSH(`x`)`	⋆
$tanh_F(x)$	`TANH(`x`)`	⋆
$coth_F(x)$	`COTH(`x`)`	†
$sech_F(x)$	`SECH(`x`)`	†
$csch_F(x)$	`CSCH(`x`)`	†
$arcsinh_F(x)$	`ASINH(`x`)`	†
$arccosh_F(x)$	`ACOSH(`x`)`	†
$arctanh_F(x)$	`ATANH(`x`)`	†
$arccoth_F(x)$	`ACOTH(`x`)`	†
$arcsech_F(x)$	`ASECH(`x`)`	†
$arccsch_F(x)$	`ACSCH(`x`)`	†

where b, x, y, u, and w are expressions of type `REAL(`***kind***`)`, and z is an expression of type `INTEGER(`***kindi***`)`.

Arithmetic value conversions in Fortran are always explicit, and the conversion function is named like the target type, except when converting to/from string formats.

$convert_{I \rightarrow I'}(x)$		`INT(`x`,`$kindi2$`)`	$\star$
	lbl_a	`FORMAT (B`n`)`	$\star$(binary)
$convert_{I'' \rightarrow I}(f)$		`READ (UNIT=#`f`,FMT=`lbl_a`)` r	$\star$
$convert_{I \rightarrow I''}(x)$		`WRITE (UNIT=#`h`, FMT=`lbl_a`)` x	$\star$
	lbl_b	`FORMAT (O`n`)`	$\star$(octal)
$convert_{I'' \rightarrow I}(f)$		`READ (UNIT=#`f`,FMT=`lbl_b`)` r	$\star$
$convert_{I \rightarrow I''}(x)$		`WRITE (UNIT=#`h`, FMT=`lbl_b`)` x	$\star$
	lbl_c	`FORMAT (I`n`)`	$\star$(decimal)
$convert_{I'' \rightarrow I}(f)$		`READ (UNIT=#`f`,FMT=`lbl_c`)` r	$\star$
$convert_{I \rightarrow I''}(x)$		`WRITE (UNIT=#`h`, FMT=`lbl_c`)` x	$\star$
	lbl_d	`FORMAT (Z`n`)`	$\star$(hexadecimal)
$convert_{I'' \rightarrow I}(f)$		`READ (UNIT=#`f`,FMT=`lbl_d`)` r	$\star$
$convert_{I \rightarrow I''}(x)$		`WRITE (UNIT=#`h`, FMT=`lbl_d`)` x	$\star$
$floor_{F \rightarrow I}(y)$		`FLOOR(`y`, `$kindi?$`)`	$\star$
$rounding_{F \rightarrow I}(y)$		`ROUND(`y`, `$kindi?$`)`	$\dagger$
$ceiling_{F \rightarrow I}(y)$		`CEILING(`y`, `$kindi?$`)`	$\star$
$convert_{I \rightarrow F}(x)$		`REAL(`x`, `$kind$`)` or sometimes `DBLE(`x`)`	$\star$
$convert_{F \rightarrow F'}(y)$		`REAL(`y`, `$kind2$`)` or sometimes `DBLE(`y`)`	$\star$
	lbl_e	`FORMAT (F`w`.`d`)`	$\star$
	lbl_f	`FORMAT (D`w`.`d`)`	$\star$
	lbl_g	`FORMAT (E`w`.`d`)`	$\star$
	lbl_h	`FORMAT (E`w`.`d`E`e`)`	$\star$
	lbl_i	`FORMAT (EN`w`.`d`)`	$\star$
	lbl_j	`FORMAT (EN`w`.`d`E`e`)`	$\star$
	lbl_k	`FORMAT (ES`w`.`d`)`	$\star$
	lbl_l	`FORMAT (ES`w`.`d`E`e`)`	$\star$
$convert_{F'' \rightarrow F}(f)$		`READ (UNIT=#`f`, FMT=`lbl_x`)` t	$\star$
$convert_{F \rightarrow F''}(y)$		`WRITE (UNIT=#`h`, FMT=`lbl_x`)` y	$\star$
$convert_{D' \rightarrow F}(f)$		`READ (UNIT=#`f`, FMT=`lbl_x`)` t	$\star$

where x is an expression of type `INTEGER(`***kindi***`)`, y is an expression of type `REAL(`***kind***`)`, f is an input file with unit number `#`f, and h is an output file with unit number `#`h. w, d, and e are literal digit (0-9) sequences, giving total, decimals, and exponent widths. lbl_x is one of lbl_e to lbl_l; all of the $lbl_$s are labels for formats.

Fortran provides base 10 non-negative numerals for all of its integer and floating point types. Numerals for floating point types must have a '.' in them. The details are not repeated in this example binding, see ISO/IEC 1539-1:1997, clause 4.3.1.1 Integer type, and clause 4.3.1.2 Real type.

Fortran does not specify numerals for infinities and NaNs. Suggestion:

$+\infty$	`INFINITY`	†
qNaN	`NAN`	†
sNaN	`NANSIGNALLING`	†

as well as string formats for reading and writing these values as character strings.

Fortran implementations can provide recording of indicators for floating point arithmetic notifications, the LIA-2 preferred method. See ISO/IEC TR 15580:1998, *Information technology – Programming languages – Fortran – Floating-point exception handling* [23]. **absolute_precision_underflow** notifications are however ignored.

C.6 Haskell

The programming language Haskell is defined by *Report on the programming language Haskell 98* [65], together with *Standard libraries for the Haskell 98 programming language* [66].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

The Haskell datatype `Bool` corresponds to the LIA datatype **Boolean**.

Every implementation of Haskell has at least two integer datatypes, `Integer`, which is unbounded, and `Int`, and at least two floating point datatypes, `Float`, and `Double`. The notation *INT* is used to stand for the name of one of the integer datatypes, and *FLT* is used to stand for the name of one of the floating point datatypes in what follows.

Haskell has a type class system that allows for overloading, and allowing static type checking of dynamic overloading. But in contrast to object oriented programming languages, type classes are not types. E.g. + has the type `(Num a) => a -> a -> a`, where `Num` is a type class and `a` is a type variable.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	`max` x y or x `‘max‘` y	⋆
$min_I(x, y)$	`min` x y or x `‘min‘` y	⋆
$max\_seq_I(xs)$	`maximum` xs	⋆
$min\_seq_I(xs)$	`minimum` xs	⋆
$dim_I(x, y)$	`dim` x y or x `‘dim‘` y	†
$power_I(x, y)$	x `^` y or `(^)` x y	⋆
$shift2_I(x, y)$	`shift2` x y or x `‘shift2‘` y	†
$shift10_I(x, y)$	`shift10` x y or x `‘shift10‘` y	†
$sqrt_I(x)$	`isqrt` x	†
$divides_I(x, y)$	`divides` x y or x `‘divides‘` y	†
$even_I(x)$	`even` x	⋆
$odd_I(x)$	`odd` x	⋆
$quot_I(x, y)$	`div` x y or x `‘div‘` y	⋆
$mod_I(x, y)$	`mod` x y or x `‘mod‘` y	⋆

$\mathit{ratio}_I(x, y)$	ratio x y or x ‘ratio‘ y	†
$\mathit{residue}_I(x, y)$	residue x y or x ‘residue‘ y	†
$\mathit{group}_I(x, y)$	grp x y or x ‘grp‘ y	†
$\mathit{pad}_I(x, y)$	pad x y or x ‘pad‘ y	†
$\mathit{gcd}_I(x, y)$	gcd x y or x ‘gcd‘ y	⋆
$\mathit{lcm}_I(x, y)$	lcm x y or x ‘lcm‘ y	⋆
$\mathit{gcd\_seq}_I(xs)$	gcd_seq xs	†
$\mathit{lcm\_seq}_I(xs)$	lcm_seq xs	†
$\mathit{add\_wrap}_I(x, y)$	x +: y	†
$\mathit{add\_ov}_I(x, y)$	x +:+ y	†
$\mathit{sub\_wrap}_I(x, y)$	x -: y	†
$\mathit{sub\_ov}_I(x, y)$	x -:+ y	†
$\mathit{mul\_wrap}_I(x, y)$	x *: y	†
$\mathit{mul\_ov}_I(x, y)$	x *:+ y	†

where x and y are expressions of type *INT* and where xs is an expression of type [*INT*].

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$\mathit{max}_F(x, y)$	max x y or x ‘max‘ y	⋆
$\mathit{min}_F(x, y)$	min x y or x ‘min‘ y	⋆
$\mathit{mmax}_F(x, y)$	mmax x y or x ‘mmax‘ y	†
$\mathit{mmin}_F(x, y)$	mmin x y or x ‘mmin‘ y	†
$\mathit{max\_seq}_F(xs)$	maximum xs	⋆
$\mathit{min\_seq}_F(xs)$	minimum xs	⋆
$\mathit{mmax\_seq}_F(xs)$	mmaximum xs	†
$\mathit{mmin\_seq}_F(xs)$	mminimum xs	†
$\mathit{dim}_F(x, y)$	dim x y or x ‘dim‘ y	†
$\mathit{floor}_F(x)$	fromInteger (floor x)	⋆
$\mathit{floor\_rest}_F(x)$	x - fromInteger (floor x)	⋆
$\mathit{rounding}_F(x)$	fromInteger (round x)	⋆
$\mathit{rounding\_rest}_F(x)$	x - fromInteger (round x)	⋆
$\mathit{ceiling}_F(x)$	fromInteger (ceiling x)	⋆
$\mathit{ceiling\_rest}_F(x)$	x - fromInteger (ceiling x)	⋆
$\mathit{residue}_F(x, y)$	residue x y or x ‘residue‘ y	†
$\mathit{sqrt}_F(x)$	sqrt x	⋆
$\mathit{rec\_sqrt}_F(x)$	rec_sqrt x	†
$\mathit{mul}_{F \to F'}(x, y)$	prod x y	†
$\mathit{add\_lo}_F(x, y)$	x +:- y	†
$\mathit{sub\_lo}_F(x, y)$	x -:- y	†
$\mathit{mul\_lo}_F(x, y)$	x *:- y	†
$\mathit{div\_rest}_F(x, y)$	x /:* y	†
$\mathit{sqrt\_rest}_F(x)$	sqrt_rest x	†

where x and y are expressions of type *FLT*, and where xs is an expression of type [*FLT*].

The binding for the floor, round, and ceiling operations here take advantage of the unbounded `Integer` type in Haskell, and that `Integer` is the default integer type.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`err_hypotenuse` x	†
$max\_error\_exp_F$	`err_exp` x	†
$max\_error\_power_F$	`err_power` x	†
$big\_angle\_r_F$	`big_radian_angle` x	†
$max\_error\_rad_F$	`err_rad` x	†
$max\_error\_sin_F$	`err_sin` x	†
$max\_error\_tan_F$	`err_tan` x	†
$min\_angular\_unit_F$	`min_angle_unit` x	†
$big\_angle\_u_F$	`big_angle` x	†
$max\_error\_sinu_F(u)$	`err_sin_cycle` u	†
$max\_error\_tanu_F(u)$	`err_tan_cycle` u	†
$max\_error\_sinh_F$	`err_sinh` x	†
$max\_error\_tanh_F$	`err_tanh` x	†
$max\_error\_convert_F$	`err_convert` x	†
$max\_error\_convert_{F'}$	`err_convert ""`	†
$max\_error\_convert_{D'}$	`err_convert ""`	†

where x and u are expressions of type *FLT*. Several of the parameter functions are constant for each type (and library), the argument is then used only to differentiate among the floating point types.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`hypotenuse` x y	†
$power_{F,I}(b, z)$	b `^^` z or `(^^)` b z	⋆
$exp_F(x)$	`exp` x	⋆
$expm1_F(x)$	`expM1` x	†
$exp2_F(x)$	`exp2` x	†
$exp10_F(x)$	`exp10` x	†
$power_F(b, y)$	b `**` y or `(**)` b y	⋆
$power1pm1_F(b, y)$	`power1PM1` b y or b `‘power1PM1‘` y	†
$ln_F(x)$	`log` x	⋆
$ln1p_F(x)$	`log1P` x	†
$log2_F(x)$	`log2` x	†
$log10_F(x)$	`log10` x	†
$logbase_F(b, x)$	`logBase` b x or b `‘logBase‘` x	⋆
$logbase1p1p_F(b, x)$	`logBase1P1P` b x	†

$rad_F(x)$	`radians` x	†
$axis\_rad_F(x)$	`axis_radians` x	†
$sin_F(x)$	`sin` x	⋆
$cos_F(x)$	`cos` x	⋆
$tan_F(x)$	`tan` x	⋆
$cot_F(x)$	`cot` x	†
$sec_F(x)$	`sec` x	†
$csc_F(x)$	`csc` x	†
$cossin_F(x)$	`cosSin` x	†
$arcsin_F(x)$	`asin` x	⋆
$arccos_F(x)$	`acos` x	⋆
$arctan_F(x)$	`atan` x	⋆
$arccot_F(x)$	`acot` x	†
$arccotc_F(x)$	`acotc` x	†
$arcsec_F(x)$	`asec` x	†
$arccsc_F(x)$	`acsc` x	†
$arc_F(x, y)$	`atan2` y x	⋆
$cycle_F(u, x)$	`cycle` u x	†
$axis\_cycle_F(u, x)$	`axis_cycle` u x	†
$sinu_F(u, x)$	`sinU` u x	†
$cosu_F(u, x)$	`cosU` u x	†
$tanu_F(u, x)$	`tanU` u x	†
$cotu_F(u, x)$	`cotU` u x	†
$secu_F(u, x)$	`secU` u x	†
$cscu_F(u, x)$	`cscU` u x	†
$cossinu_F(u, x)$	`cosSinU` u x	†
$arcsinu_F(u, x)$	`asinU` u x	†
$arccosu_F(u, x)$	`acosU` u x	†
$arctanu_F(u, x)$	`atanU` u x	†
$arccotu_F(u, x)$	`acotU` u x	†
$arccotcu_F(u, x)$	`acotcU` u x	†
$arcsecu_F(u, x)$	`asecU` u x	†
$arccscu_F(u, x)$	`acscU` u x	†
$arcu_F(u, x, y)$	`atan2U` u y x	†
$rad\_to\_cycle_F(x, w)$	`rad_to_cycle` w x	†
$cycle\_to\_rad_F(u, x)$	`cycle_to_rad` u x	†
$cycle\_to\_cycle_F(u, x, w)$	`cycle_to_cycle` u x w	†
$sinh_F(x)$	`sinh` x	⋆
$cosh_F(x)$	`cosh` x	⋆
$tanh_F(x)$	`tanh` x	⋆

$coth_F(x)$	`coth` x	†
$sech_F(x)$	`sech` x	†
$csch_F(x)$	`csch` x	†
$arcsinh_F(x)$	`asinh` x	⋆
$arccosh_F(x)$	`acosh` x	⋆
$arctanh_F(x)$	`atanh` x	⋆
$arccoth_F(x)$	`acoth` x	†
$arcsech_F(x)$	`asech` x	†
$arccsch_F(x)$	`acsch` x	†

where b, x, y, u, and w are expressions of type *FLT*, and z is an expression of type *INT*.

Arithmetic value conversions in Haskell are always explicit. They are done with the overloaded `fromIntegral` and `fromFractional` operations.

$convert_{I \to I'}(x)$	`fromIntegral` x	⋆
$convert_{I'' \to I}(x)$	`read` s	⋆
$convert_{I \to I''}(x)$	`show` x	⋆
$floor_{F \to I}(y)$	`floor` y	⋆
$rounding_{F \to I}(y)$	`round` y	⋆
$ceiling_{F \to I}(y)$	`ceiling` y	⋆
$convert_{I \to F}(x)$	`fromIntegral` x	⋆
$convert_{F \to F'}(y)$	`fromFractional` y	⋆
$convert_{F'' \to F}(s)$	`read` s	⋆
$convert_{F \to F''}(y)$	`show` y	⋆
$convert_{D' \to F}(s)$	`read` s	⋆
$convert_{F \to D'}(y)$	`show` y	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*. `show` does not allow for format control.

Haskell provides non-negative numerals for all its integer and floating point types in base 10. There is no differentiation between the numerals for different floating point datatypes, nor between numerals for different integer datatypes, and integer numerals can be used for floating point values. Integer numerals stand for a value in `Integer` (the unbounded integer type) and an implicit `fromInteger` operation is applied to it. Fractional numerals stand for a value in `Rational` (the unbounded type of rational numbers) and an implicit `fromRational` operation is applied to it.

Haskell does not specify any numerals for infinities and NaNs. Suggestion:

$+\infty$	`infinity`	†
qNaN	`quietNaN`	†
sNaN	`sigallingNaN`	†

as well as string formats for reading and writing these values as character strings.

Haskell has the notion of `error`, which results in a change of 'control flow', which cannot be returned from, nor caught. An `error` results in the termination of the program. (There are suggestions to improve this.) **infinitary** for integer types and **invalid** (in general) are considered to be `error`. No notification results for **underflow**, and the continuation value (specified by LIA-2) is used directly, since recording of indicators is not available and `error` is inappropriate for **underflow**. The handling of integer **overflow** is implementation dependent. The handling of floating point **overflow** and **infinitary** should be to return a suitable infinity (specified by LIA-2), if possible, without any notification, since recording of indicators is not available.

C.7 Java

The programming language Java is defined by *The Java Language Specification* [64], plus a number of class libraries (exactly which vary depending on the Java 'edition' and version).

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided. The LIA-2 operations that are provided in Java 2 (marked "⋆" below) are in the final class `java.lang.Math`.

The Java datatype `boolean` corresponds to the LIA datatype **Boolean**.

Every implementation of Java has the integer datatypes `int` and `long`. The notation *INT* will be used to refer to either one of them.

Java has two floating point datatypes, `float` and `double`, which must conform to IEC 60559. The notation *FLT* will be used to refer to either one of them.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$min_I(x, y)$	`min(`x`, `y`)`	⋆
$max_I(x, y)$	`max(`x`, `y`)`	⋆
$min\_seq_I(xs)$	`min_arr(`xs`)`	†
$max\_seq_I(xs)$	`max_arr(`xs`)`	†
$dim_I(x, y)$	`dim(`x`, `y`)`	†
$power_I(x, y)$	`power(`x`, `y`)`	†
$shift2_I(x, y)$	`shift2(`x`, `y`)`	†
$shift10_I(x, y)$	`shift10(`x`, `y`)`	†
$sqrt_I(x)$	`sqrt(`x`)`	†
$divides_I(x, y)$	`divides(`x`, `y`)`	†
$divides_I(x, y)$	x `!= 0 &&` y `%` x `== 0`	⋆
$even_I(x)$	x `% 2 == 0`	⋆
$odd_I(x)$	x `% 2 != 0`	⋆
$quot_I(x, y)$	`quot(`x`, `y`)`	†
$mod_I(x, y)$	`mod(`x`, `y`)`	†
$ratio_I(x, y)$	`ratio(`x`, `y`)`	†
$residue_I(x, y)$	`residue(`x`, `y`)`	†

$group_I(x, y)$	`group(`x`, `y`)`	†
$pad_I(x, y)$	`pad(`x`, `y`)`	†
$gcd_I(x, y)$	`gcd(`x`, `y`)`	†
$lcm_I(x, y)$	`lcm(`x`, `y`)`	†
$gcd\_seq_I(xs)$	`gcd_arr(`xs`)`	†
$lcm\_seq_I(xs)$	`lcm_arr(`xs`)`	†
$add\_wrap_I(x, y)$	`add_wrap(`x`, `y`)`	†
$add\_ov_I(x, y)$	`add_over(`x`, `y`)`	†
$sub\_wrap_I(x, y)$	`sub_wrap(`x`, `y`)`	†
$sub\_ov_I(x, y)$	`sub_over(`x`, `y`)`	†
$mul\_wrap_I(x, y)$	`mul_wrap(`x`, `y`)`	†
$mul\_ov_I(x, y)$	`mul_over(`x`, `y`)`	†

where x and y are expressions of type *INT* and where xs is an expression of type *INT*`[]`.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$min_F(x, y)$	`min(`x`, `y`)`	⋆
$max_F(x, y)$	`max(`x`, `y`)`	⋆
$mmax_F(x, y)$	`mmax(`x`, `y`)`	†
$mmin_F(x, y)$	`mmin(`x`, `y`)`	†
$min\_seq_F(xs)$	`min_arr(`xs`)`	†
$max\_seq_F(xs)$	`max_arr(`xs`)`	†
$mmax\_seq_F(xs)$	`mmax(`xs`)`	†
$mmin\_seq_F(xs)$	`mmin(`xs`)`	†
$dim_F(x, y)$	`dim(`x`, `y`)`	†
$floor_F(x)$	`floor(`x`)`	⋆(only for `double`)
$floor\_rest_F(x)$	x `- floor(`x`)`	⋆(only for `double`)
$rounding_F(x)$	`rint(`x`)`	⋆(only for `double`)
$rounding\_rest_F(x)$	x `- rint(`x`)`	⋆(only for `double`)
$ceiling_F(x)$	`ceil(`x`)`	⋆(only for `double`)
$ceiling\_rest_F(x)$	x `- ceil(`x`)`	⋆(only for `double`)
$residue_F(x, y)$	`IEEEremainder(`x`, `y`)`	⋆ (only for `double`)
$sqrt_F(x)$	`sqrt(`x`)`	⋆
$rec\_sqrt_F(x)$	`rec_sqrt(`x`)`	†
$mul_{F \rightarrow F'}(x, y)$	`dprod(`x`, `y`)`	†
$add\_lo_F(x, y)$	`add_low(`x`, `y`)`	†
$sub\_lo_F(x, y)$	`sub_low(`x`, `y`)`	†
$mul\_lo_F(x, y)$	`mul_low(`x`, `y`)`	†
$div\_rest_F(x, y)$	`div_rest(`x`, `y`)`	†
$sqrt\_rest_F(x)$	`sqrt_rest(`x`)`	†

where x and y are expressions of type *FLT*, and where xs is an expression of type *FLT*`[]`.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`err_hypotenuse(`x`)`	†
$max\_error\_exp_F$	`err_exp(`x`)`	†
$max\_error\_power_F(b,x)$	`err_power(`x`)`	†
$big\_angle\_r_F$	`big_radian_angle(`x`)`	†
$max\_error\_rad_F$	`err_rad(`x`)`	†
$max\_error\_sin_F$	`err_sin(`x`)`	†
$max\_error\_tan_F$	`err_tan(`x`)`	†
$min\_angular\_unit_F$	`smallest_angular_unit(`x`)`	†
$big\_angle\_u_F$	`big_angle(`x`)`	†
$max\_error\_sinu_F(u)$	`err_sin_cycle(`u`)`	†
$max\_error\_tanu_F(u)$	`err_tan_cycle(`u`)`	†
$max\_error\_sinh_F$	`err_sinh(`x`)`	†
$max\_error\_tanh_F$	`err_tanh(`x`)`	†
$max\_error\_convert_F$	`err_convert(`x`)`	†
$max\_error\_convert_{F'}$	`err_convert_to_string`	†
$max\_error\_convert_{D'}$	`err_convert_to_string`	†

where x and u are expressions of type *FLT*. Several of the parameter functions are constant for each type (and library), the argument is then used only to differentiate among the floating point types.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them. These are defined only for `double` not for `float`.

$hypot_F(x,y)$	`hypotenuse(`x`, `y`)`	†
$power_{F,I}(b,z)$	`poweri(`b`, `z`)`	†
$exp_F(x)$	`exp(`x`)`	⋆
$expm1_F(x)$	`expm1(`x`)`	†
$exp2_F(x)$	`exp2(`x`)`	†
$exp10_F(x)$	`exp10(`x`)`	†
$power_F(b,y)$	`power(`b`, `y`)`	†
$pow_F(b,y)$	`pow(`b`, `y`)`	⋆ Not LIA-2!
$power1pm1_F(b,y)$	`power1pm1(`b`, `y`)`	†
$ln_F(x)$	`log(`x`)`	⋆
$ln1p_F(x)$	`log1p(`x`)`	†
$log2_F(x)$	`log2(`x`)`	†
$log10_F(x)$	`log10(`x`)`	†
$logbase_F(b,x)$	`log(`b`, `x`)`	†
$logbase1p1p_F(b,x)$	`log1p1p(`b`, `x`)`	†
$rad_F(x)$	`radian(`x`)`	†
$axis\_rad_F(x)$	`axis_rad(`x`)`	†

$sin_F(x)$	`sin(`x`)`	⋆
$cos_F(x)$	`cos(`x`)`	⋆
$tan_F(x)$	`tan(`x`)`	⋆
$cot_F(x)$	`cot(`x`)`	†
$sec_F(x)$	`sec(`x`)`	†
$csc_F(x)$	`csc(`x`)`	†
$arcsin_F(x)$	`asin(`x`)`	⋆
$arccos_F(x)$	`acos(`x`)`	⋆
$arctan_F(x)$	`atan(`x`)`	⋆
$arccot_F(x)$	`acot(`x`)`	†
$arccotc_F(x)$	`acotc(`x`)`	†
$arcsec_F(x)$	`asec(`x`)`	†
$arccsc_F(x)$	`acsc(`x`)`	†
$arc_F(x, y)$	`atan2(`y`, `x`)`	⋆
$cycle_F(u, x)$	`cycle(`u`, `x`)`	†
$axis\_cycle_F(u, x)$	`axis_cycle(`u`, `x`)`	†
$sinu_F(u, x)$	`sinu(`u`, `x`)`	†
$cosu_F(u, x)$	`cosu(`u`, `x`)`	†
$tanu_F(u, x)$	`tanu(`u`, `x`)`	†
$cotu_F(u, x)$	`cotu(`u`, `x`)`	†
$secu_F(u, x)$	`secu(`u`, `x`)`	†
$cscu_F(u, x)$	`cscu(`u`, `x`)`	†
$arcsinu_F(u, x)$	`asinu(`u`, `x`)`	†
$arccosu_F(u, x)$	`acosu(`u`, `x`)`	†
$arctanu_F(u, x)$	`atanu(`u`, `x`)`	†
$arccotu_F(u, x)$	`acotu(`u`, `x`)`	†
$arccotcu_F(u, x)$	`acotcu(`u`, `x`)`	†
$arcsecu_F(u, x)$	`asecu(`u`, `x`)`	†
$arccscu_F(u, x)$	`acscu(`u`, `x`)`	†
$arcu_F(u, x, y)$	`atan2u(`u`, `y`, `x`)`	†
$rad\_to\_cycle_F(x, 360)$	`toDegrees(`x`)`	⋆
$cycle\_to\_rad_F(360, x)$	`toRadians(`x`)`	⋆
$rad\_to\_cycle_F(x, w)$	`radian_to_cycle(`x`, `w`)`	†
$cycle\_to\_rad_F(u, x)$	`cycle_to_radian(`u`, `x`)`	†
$cycle\_to\_cycle_F(u, x, w)$	`cycle_to_cycle(`u`, `x`, `w`)`	†
$sinh_F(x)$	`sinh(`x`)`	†
$cosh_F(x)$	`cosh(`x`)`	†
$tanh_F(x)$	`tanh(`x`)`	†
$coth_F(x)$	`coth(`x`)`	†
$sech_F(x)$	`sech(`x`)`	†
$csch_F(x)$	`csch(`x`)`	†

$arcsinh_F(x)$	`asinh(`x`)`	†
$arccosh_F(x)$	`acosh(`x`)`	†
$arctanh_F(x)$	`atanh(`x`)`	†
$arccoth_F(x)$	`acoth(`x`)`	†
$arcsech_F(x)$	`asech(`x`)`	†
$arccsch_F(x)$	`acsch(`x`)`	†

where b, x, y, u, and w are expressions of type *FLT*, and z is an expression of type *INT*.

Arithmetic value conversions in Java can be explicit or implicit. The rules for when implicit conversions are applied is not repeated here. The explicit arithmetic value conversions are usually expressed as 'casts', except when converting to/from strings.

$convert_{I \to I'}(x)$	`(`*INT2*`)`x	⋆
$convert_{I'' \to I}(s)$	`Integer.parseInt(`s`)`	⋆
$convert_{I'' \to I}(s)$	`Integer.parseInt(`s`, `*radix*`)`	⋆
$convert_{I'' \to I}(s)$	`Long.parseLong(`s`)`	⋆
$convert_{I'' \to I}(s)$	`Long.parseLong(`s`, `*radix*`)`	⋆
$convert_{I \to I''}(x)$	`Integer.toString(`x`)`	⋆
$convert_{I \to I''}(x)$	`Integer.toString(`x`, `*radix*`)`	⋆
$convert_{I \to I''}(x)$	`Integer.toBinaryString(`x`)`	⋆
$convert_{I \to I''}(x)$	`Integer.toOctalString(`x`)`	⋆
$convert_{I \to I''}(x)$	`Integer.toHexString(`x`)`	⋆
$convert_{I \to I''}(x)$	`Long.toString(`x`)`	⋆
$convert_{I \to I''}(x)$	`Long.toString(`x`, `*radix*`)`	⋆
$convert_{I \to I''}(x)$	`Long.toBinaryString(`x`)`	⋆
$convert_{I \to I''}(x)$	`Long.toOctalString(`x`)`	⋆
$convert_{I \to I''}(x)$	`Long.toHexString(`x`)`	⋆
$convert_{I \to I''}(x)$	`""+`x	⋆
$floor_{F \to I}(y)$	`(`*INT*`)floor(`y`)`	⋆
$rounding_{F \to I}(y)$	`(`*INT*`)rint(`y`)`	⋆
$ceiling_{F \to I}(y)$	`(`*INT*`)ceil(`y`)`	⋆
$convert_{I \to F}(x)$	`(`*FLT*`)`x	⋆
$convert_{F \to F'}(y)$	`(`*FLT2*`)`y	⋆
$convert_{F'' \to F}(s)$	`Float.parseFloat(`s`)`	⋆
$convert_{F'' \to F}(s)$	`Double.parseDouble(`s`)`	⋆
$convert_{F \to F''}(y)$	`Float.toString(`y`)`	⋆
$convert_{F \to F''}(y)$	`Double.toString(`y`)`	⋆
$convert_{D' \to F}(s)$	`Float.parseFloat(`s`)`	⋆
$convert_{D' \to F}(s)$	`Double.parseDouble(`s`)`	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*, and s is an expression of type `String`. `toString` does not provide any format control. *INT2* is the integer datatype that corresponds to I'. *FLT2* is the floating point datatype that corresponds to F'.

Java provides non-negative numerals for all its integer and floating point types. The default base is 10, but for integers base 8 and 16 can be used too. Numerals for different integer types are distinguished by suffixes. Numerals for different floating point types are distinguished by suffix: `f` for `float`, no suffix for `double`, `l` for `long double`. Numerals for floating point types must have a '.' in them. The details are not repeated in this example binding, see *The Java Language Specification*, clause 3.10.1 Integer literals, and clause 3.10.2 Floating-point literals.

Java specifies numerals for infinities and NaNs:

$+\infty$	`Float.POSITIVE_INFINITY`	⋆
$+\infty$	`Double.POSITIVE_INFINITY`	⋆
$-\infty$	`Float.NEGATIVE_INFINITY`	⋆
$-\infty$	`Double.NEGATIVE_INFINITY`	⋆
qNaN	`Float.NaN`	⋆
qNaN	`Double.NaN`	⋆
sNaN	`Float.SigNaN`	†
sNaN	`Double.SigNaN`	†

as well as string formats for writing these values as character strings. However, infinities and NaNs cannot be converted *from* string.

Java has a notion of 'exception' that implies a non-returnable, but catchable, change of control flow. Java uses its exception mechanism as its default means of notification. **underflow** does not cause any notification in Java, and the continuation value to the **underflow** is used directly, since an Java exception is inappropriate for an **underflow** notification. Java also ignores **infinitary** and **overflow** notifications for floating point operations, and the continuation value (specified in LIA-2) is used directly without recording the **infinitary** or **overflow** itself. Java uses the exception `java.lang.ArithmeticException` for **invalid** notifications and for **infinitary** notifications for integer operations. Java uses `java.lang.NumberFormatException` for **invalid** notifications for operations that convert from string. Since Java exceptions are non-returnable changes of control flow, no continuation value is provided for these notifications.

An implementation that wishes to follow LIA-2 should provide recording of indicators as an alternative means of handling numeric notifications, including those that Java ignores when the numeric notification handling mechanism is by Java exceptions. Recording of indicators is the LIA-2 preferred means of handling numeric notifications.

C.8 Common Lisp

The programming language Common Lisp is defined by ANSI X3.226-1994, *Information Technology – Programming Language – Common Lisp* [42].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

Common Lisp does not have a single datatype that corresponds to the LIA-1 datatype **Boolean**. Rather, `NIL` corresponds to **false** and `T` corresponds to **true**.

Every implementation of Common Lisp has one unbounded integer datatype. Any mathematical integer is assumed to have a representation as a Common Lisp data object, subject only to total memory limitations.

Common Lisp has four floating point datatypes: `short-float`, `single-float`, `double-float`, and `long-float`. Not all of these floating point datatypes must be distinct.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	`(max` x y`)`	⋆
$min_I(x, y)$	`(min` x y`)`	⋆
$max\_seq_I(xs)$	`(max .` xs`)` or `(max` x_1 x_2 `...` x_n`)`	⋆
$min\_seq_I(xs)$	`(min .` xs`)` or `(min` x_1 x_2 `...` x_n`)`	⋆
$dim_I(x, y)$	`(dim` x y`)`	†
$power_I(x, y)$	`(expt` x y`)` (returns a rational on negative power)	⋆
$shift2_I(x, y)$	`(shift2` x y`)`	†
$shift10_I(x, y)$	`(shift10` x y`)`	†
$sqrt_I(x)$	`(isqrt` x`)`	⋆
$divides_I(x, y)$	`(dividesp` x y`)`	†
$even_I(x)$	`(evenp` x`)`	⋆
$odd_I(x)$	`(oddp` x`)`	⋆
	(the `floor`, `ceiling`, and `round` can also accept floating point arguments)	
	`(multiple-value-bind (flr md) (floor` x y`))`	⋆
$quot_I(x, y)$	`flr` or `(floor` x y`)`	⋆
$mod_I(x, y)$	`md` or `(mod` x y`)`	⋆
	`(multiple-value-bind (rnd rm) (round` x y`))`	⋆
$ratio_I(x, y)$	`rnd` or `(round` x y`)`	⋆
$residue_I(x, y)$	`rm`	
	`(multiple-value-bind (ceil pd) (ceiling` x y`))`	⋆
$group_I(x, y)$	`ceil` or `(ceiling` x y`)`	⋆
$pad_I(x, y)$	`(- pd)`	
$gcd_I(x, y)$	`(gcd` x y`)` (deviation: `(gcd 0 0)` is 0)	⋆
$lcm_I(x, y)$	`(lcm` x y`)`	⋆
$gcd\_seq_I(xs)$	`(gcd .` xs`)` or `(gcd` x_1 x_2 `...` x_n`)`	⋆(dev. as above)
$lcm\_seq_I(xs)$	`(lcm .` xs`)` or `(lcm` x_1 x_2 `...` x_n`)`	⋆
$add\_wrap_I(x, y)$	`(add-wrap` x y`)`	†
$add\_ov_I(x, y)$	`(add-over` x y`)`	†
$sub\_wrap_I(x, y)$	`(sub-wrap` x y`)`	†
$sub\_ov_I(x, y)$	`(sub-over` x y`)`	†
$mul\_wrap_I(x, y)$	`(mul-wrap` x y`)`	†
$mul\_ov_I(x, y)$	`(mul-over` x y`)`	†

where x and y are expressions of type *INT* and where xs is an expression of type list of *INT*.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	`(max` x y`)`	⋆
$min_F(x, y)$	`(min` x y`)`	⋆
$mmax_F(x, y)$	`(mmax` x y`)`	†

$mmin_F(x, y)$	`(mmin x y)`	†
$max\_seq_F(xs)$	`(max . xs)` or `(max x1 x2 ... xn)`	⋆
$min\_seq_F(xs)$	`(min . xs)` or `(min x1 x2 ... xn)`	⋆
$mmax\_seq_F(xs)$	`(mmax . xs)` or `(mmax x1 x2 ... xn)`	†
$mmin\_seq_F(xs)$	`(mmin . xs)` or `(mmin x1 x2 ... xn)`	†
$dim_F(x, y)$	`(dim x y)`	†
`(multiple-value-bind (flr frem) (ffloor x))`		⋆
$floor_F(x)$	`(ffloor x)` or `flr`	⋆
$floor\_rest_F(x)$	`frem`	
`(multiple-value-bind (rnd rrem) (fround x))`		⋆
$rounding_F(x)$	`(fround x)` or `rnd`	⋆
$rounding\_rest_F(x)$	`rrem`	
`(multiple-value-bind (cln crem) (fceiling x))`		⋆
$ceiling_F(x)$	`(fceiling x)` or `cln`	⋆
$ceiling\_rest_F(x)$	`crem`	
`(multiple-value-bind (rnd rm) (fround x y))`		⋆
$residue_F(x, y)$	`rm`	
$sqrt_F(x)$	`(sqrt x)` (returns a complex when $x < 0$)	⋆
$rec\_sqrt_F(x)$	`(rec-sqrt x)`	†
$mul_{F \to F'}(x, y)$	`(prod x y)`	†
$add\_lo_F(x, y)$	`(add-low x y)`	†
$sub\_lo_F(x, y)$	`(sub-low x y)`	†
$mul\_lo_F(x, y)$	`(mul-low x y)`	†
$div\_rest_F(x, y)$	`(div-rest x y)`	†
$sqrt\_rest_F(x)$	`(sqrt-rest x)`	†

where x and y are data objects of the same floating point type, and where xs is a data object that is a list of data objects of (the same, in this binding) floating point type. Note that Common Lisp allows mixed number datatypes in many of its operations. This example binding does not explain that in detail.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`(err-hypotenuse x)`	†
$max\_error\_exp_F$	`(err-exp x)`	†
$max\_error\_power_F$	`(err-power x)`	†
$big\_angle\_r_F$	`(big-radian-angle x)`	†
$max\_error\_rad_F$	`(err-rad x)`	†
$max\_error\_sin_F$	`(err-sin x)`	†
$max\_error\_tan_F$	`(err-tan x)`	†
$min\_angular\_unit_F$	`(minimum-angular-unit x)`	†

$big\_angle\_u_F$	`(big-angle` x`)`	†
$max\_error\_sinu_F(u)$	`(err-sin-cycle` u`)`	†
$max\_error\_tanu_F(u)$	`(err-tan-cycle` u`)`	†
$max\_error\_sinh_F$	`(err-sinh` x`)`	†
$max\_error\_tanh_F$	`(err-tanh` x`)`	†
$max\_error\_convert_F$	`(err-convert` x`)`	†
$max\_error\_convert_{F'}$	`err-convert-to-string`	†
$max\_error\_convert_{D'}$	`err-convert-to-string`	†

where x and u are expressions of type *FLT*. Several of the parameter functions are constant for each type (and library), the argument is then used only to differentiate among the floating point datatypes.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`(hypotenuse` x y`)`	†
$power_{F,I}(b, z)$	`(expt` b z`)`	⋆
$exp_F(x)$	`(exp` x`)`	⋆
$exp2_F(x)$	`(exp2` x`)`	†
$exp10_F(x)$	`(exp10` x`)`	†
$expm1_F(x)$	`(expm1` x`)`	†
$power_F(b, y)$	`(expt` b y`)` (deviation: `(expt 0.0 0.0)` is 1)	⋆
$power1pm1_F(b, y)$	`(expt1pm1` b y`)`	†
$ln_F(x)$	`(log` x`)` (returns a complex on negative arg.)	⋆
$ln1p_F(x)$	`(log1p` x`)`	†
$log2_F(x)$	`(log2` x`)`	†
$log10_F(x)$	`(log10` x`)`	†
$logbase_F(b, x)$	`(log` x b`)` (note parameter order)	⋆
$logbase1p1p_F(b, x)$	`(log1p1p` x b`)`	†
$rad_F(x)$	`(radians` x`)`	†
$axis\_rad_F(x)$	`(axis-rad` x`)`	†
$sin_F(x)$	`(sin` x`)`	⋆
$cos_F(x)$	`(cos` x`)`	⋆
$tan_F(x)$	`(tan` x`)`	⋆
$cot_F(x)$	`(cot` x`)`	†
$sec_F(x)$	`(sec` x`)`	†
$csc_F(x)$	`(csc` x`)`	†
$cossin_F(x)$	`(cossin` x`)`	†
$arcsin_F(x)$	`(asin` x`)` (returns a complex when $\lvert x\rvert > 1$)	⋆
$arccos_F(x)$	`(acos` x`)` (returns a complex when $\lvert x\rvert > 1$)	⋆
$arctan_F(x)$	`(atan` x`)`	⋆
$arccot_F(x)$	`(acot` x`)`	†

$arccotc_F(x)$	`(acotc` x`)`		†
$arcsec_F(x)$	`(asec` x`)`		†
$arccsc_F(x)$	`(acsc` x`)`		†
$arc_F(x, y)$	`(atan` y x`)`		⋆
$cycle_F(u, x)$	`(cycle` u x`)`		†
$axis\_cycle_F(u, x)$	`(axis-cycle` u x`)`		†
$sinu_F(u, x)$	`(sinU` u x`)`		†
$cosu_F(u, x)$	`(cosU` u x`)`		†
$tanu_F(u, x)$	`(tanU` u x`)`		†
$cotu_F(u, x)$	`(cotU` u x`)`		†
$secu_F(u, x)$	`(secU` u x`)`		†
$cscu_F(u, x)$	`(cscU` u x`)`		†
$cossinu_F(u, x)$	`(cossinU` u x`)`		†
$arcsinu_F(u, x)$	`(asinU` u x`)`		†
$arccosu_F(u, x)$	`(acosU` u x`)`		†
$arctanu_F(u, x)$	`(atanU` u x`)`		†
$arccotu_F(u, x)$	`(acotU` u x`)`		†
$arccotcu_F(u, x)$	`(acotcU` u x`)`		†
$arcsecu_F(u, x)$	`(asecU` u x`)`		†
$arccscu_F(u, x)$	`(acscU` u x`)`		†
$arcu_F(u, x, y)$	`(atanU` u y x`)`		†
$rad\_to\_cycle_F(x, w)$	`(rad-to-cycle` x w`)`		†
$cycle\_to\_rad_F(u, x)$	`(cycle-to-rad` u x`)`		†
$cycle\_to\_cycle_F(u, x, w)$	`(cycle-to-cycle` u x w`)`		†
$sinh_F(x)$	`(sinh` x`)`		⋆
$cosh_F(x)$	`(cosh` x`)`		⋆
$tanh_F(x)$	`(tanh` x`)`		⋆
$coth_F(x)$	`(coth` x`)`		†
$sech_F(x)$	`(sech` x`)`		†
$csch_F(x)$	`(csch` x`)`		†
$arcsinh_F(x)$	`(asinh` x`)`		⋆
$arccosh_F(x)$	`(acosh` x`)`	(returns a complex when $x < 1$)	⋆
$arctanh_F(x)$	`(atanh` x`)`	(returns a complex when $\|x\| > 1$)	⋆
$arccoth_F(x)$	`(acoth` x`)`		†
$arcsech_F(x)$	`(asech` x`)`		†
$arccsch_F(x)$	`(acsch` x`)`		†

where b, x, y, u, and w are expressions of type *FLT*, and z is an expression of type *INT*.

Arithmetic value conversions in Common Lisp can be explicit or implicit. The rules for when implicit conversions are done is implementation defined.

$convert_{I \to I''}(x)$	`(format nil "~`w`B"` x`)`	⋆(binary)
$convert_{I \to I''}(x)$	`(format nil "~`w`O"` x`)`	⋆(octal)

$convert_{I \rightarrow I''}(x)$	`(format nil "~`w`D"` x`)`	⋆(decimal)
$convert_{I \rightarrow I''}(x)$	`(format nil "~`w`X"` x`)`	⋆(hexadecimal)
$convert_{I \rightarrow I''}(x)$	`(format nil "~`r, w`R"` x`)`	⋆(radix r)
$convert_{I \rightarrow I''}(x)$	`(format nil "~@R"` x`)`	⋆(roman numeral)
$floor_{F \rightarrow I}(y)$	`(floor` y`)`	⋆
$rounding_{F \rightarrow I}(y)$	`(round` y`)`	⋆
$ceiling_{F \rightarrow I}(y)$	`(ceiling` y`)`	⋆
$convert_{I \rightarrow F}(x)$	`(float` x $kind$`)`	⋆
$convert_{F \rightarrow F'}(y)$	`(float` y $kind2$`)`	⋆
$convert_{F \rightarrow F''}(y)$	`(format nil "~`w`F"` y`)`	⋆
$convert_{F \rightarrow F''}(y)$	`(format nil "~`w, e, k, c`E"` y`)`	⋆
$convert_{F \rightarrow F''}(y)$	`(format nil "~`w, e, k, c`G"` y`)`	⋆
$convert_{F \rightarrow D'}(y)$	`(format nil "~`r, w`,0,#F"` y`)`	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*. Conversion from string to numeric value is in Common Lisp done via a general read procedure, which reads Common Lisp 'S-expressions'.

Common Lisp provides non-negative numerals for all its integer and floating point datatypes in base 10. There is no differentiation between the numerals for different floating point datatypes, nor between numerals for different integer datatypes, and integer numerals can be used for floating point values.

Common Lisp does not specify numerals for infinities and NaNs. Suggestion:

$+\infty$	`infinity-`*FLT*	†
qNaN	`nan-`*FLT*	†
sNaN	`signan-`*FLT*	†

as well as string formats for reading and writing these values as character strings.

Common Lisp has a notion of 'exception'. However, Common Lisp has no notion of compile time type checking, and an operation can return differently typed values for different arguments. When justifiable, Common Lisp arithmetic operations return a rational or a complex floating point value rather than giving a notification, even if the argument(s) to the operation were not complex. For instance, `(sqrt -1)` (quietly) returns a representation of $0 + i$.

C.9 ISLisp

The programming language ISLisp is defined by ISO/IEC 13816:1997, *Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP* [24].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

ISLisp does not have a single datatype that corresponds to the LIA datatype **Boolean**. Rather, `NIL` corresponds to **false** and `T` corresponds to **true**.

Every implementation of ISLisp has one unbounded integer datatype. Any mathematical integer is assumed to have a representation as an ISLisp data object, subject only to total memory limitations.

ISLisp has one floating point type required to conform to IEC 60559.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	`(max` x y`)`	⋆
$min_I(x, y)$	`(min` x y`)`	⋆
$max\_seq_I(xs)$	`(max .` xs`)` or `(max` x_1 x_2 `...` x_n`)`	⋆
$min\_seq_I(xs)$	`(min .` xs`)` or `(min` x_1 x_2 `...` x_n`)`	⋆
$dim_I(x, y)$	`(dim` x y`)`	†
$power_I(x, y)$	`(expt` x y`)` (deviation: `(expt 0 0)` is 1)	⋆
$shift2_I(x, y)$	`(shift2` x y`)`	†
$shift10_I(x, y)$	`(shift10` x y`)`	†
$sqrt_I(x)$	`(isqrt` x`)`	⋆
$divides_I(x, y)$	`(dividesp` x y`)`	†
$even_I(x)$	`(evenp` x`)`	†
$odd_I(x)$	`(oddp` x`)`	†
$quot_I(x, y)$	`(div` x y`)`	⋆
$mod_I(x, y)$	`(mod` x y`)`	⋆
$ratio_I(x, y)$	`(ratio` x y`)`	†
$residue_I(x, y)$	`(residue` x y`)`	†
$group_I(x, y)$	`(group` x y`)`	†
$pad_I(x, y)$	`(pad` x y`)`	†
$gcd_I(x, y)$	`(gcd` x y`)` (deviation: `(gcd 0 0)` is 0)	⋆
$lcm_I(x, y)$	`(lcm` x y`)`	⋆
$gcd\_seq_I(xs)$	`(gcds` xs`)`	†
$lcm\_seq_I(xs)$	`(lcms` xs`)`	†
$add\_wrap_I(x, y)$	`(add-wrap` x y`)`	†
$add\_ov_I(x, y)$	`(add-over` x y`)`	†
$sub\_wrap_I(x, y)$	`(sub-wrap` x y`)`	†
$sub\_ov_I(x, y)$	`(sub-over` x y`)`	†
$mul\_wrap_I(x, y)$	`(mul-wrap` x y`)`	†
$mul\_ov_I(x, y)$	`(mul-over` x y`)`	†

where x and y are expressions of the unbounded integer type and where xs is an expression of type list of the unbounded integer type.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	`(max` x y`)`	⋆
$min_F(x, y)$	`(min` x y`)`	⋆

$mmax_F(x, y)$	`(mmax` x y`)`	†
$mmin_F(x, y)$	`(mmin` x y`)`	†
$max\_seq_F(xs)$	`(max .` xs`)` or `(max` x_1 x_2 `...` x_n`)`	⋆
$min\_seq_F(xs)$	`(min .` xs`)` or `(min` x_1 x_2 `...` x_n`)`	⋆
$mmax\_seq_F(xs)$	`(mmax .` xs`)` or `(mmax` x_1 x_2 `...` x_n`)`	†
$mmin\_seq_F(xs)$	`(mmin .` xs`)` or `(mmin` x_1 x_2 `...` x_n`)`	†
$dim_F(x, y)$	`(dim` x y`)`	†
$floor_F(x)$	`(float (floor` x`))`	⋆
$floor\_rest_F(x)$	`(-` x `(float (floor` x`))`	⋆
$rounding_F(x)$	`(float (round` x`))`	⋆
$rounding\_rest_F(x)$	`(-` x `(float (round` x`))`	⋆
$ceiling_F(x)$	`(float (ceiling` x`))`	⋆
$ceiling\_rest_F(x)$	`(-` x `(float (ceiling` x`))`	⋆
$residue_F(x, y)$	`(residue` x y`)`	†
$sqrt_F(x)$	`(sqrt` x`)`	⋆
$rec\_sqrt_F(x)$	`(rec-sqrt` x`)`	†
$add\_lo_F(x, y)$	`(add-low` x y`)`	†
$sub\_lo_F(x, y)$	`(sub-low` x y`)`	†
$mul\_lo_F(x, y)$	`(mul-low` x y`)`	†
$div\_rest_F(x, y)$	`(div-rest` x y`)`	†
$sqrt\_rest_F(x)$	`(sqrt-rest` x`)`	†

where x and y are data objects of the ISLisp floating point type, and where xs is a data object that is a list of data objects of the ISLisp floating point type.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`(err-hypotenuse` x`)`	†
$max\_error\_exp_F$	`(err-exp` x`)`	†
$max\_error\_power_F$	`(err-power` x`)`	†
$big\_angle\_r_F$	`(big-radian-angle` x`)`	†
$max\_error\_rad_F$	`(err-rad` x`)`	†
$max\_error\_sin_F$	`(err-sin` x`)`	†
$max\_error\_tan_F$	`(err-tan` x`)`	†
$min\_angular\_unit_F$	`(minimum-angular-unit` x`)`	†
$big\_angle\_u_F$	`(big-angle` x`)`	†
$max\_error\_sinu_F(u)$	`(err-sin-cycle` u`)`	†
$max\_error\_tanu_F(u)$	`(err-tan-cycle` u`)`	†
$max\_error\_sinh_F$	`(err-sinh` x`)`	†
$max\_error\_tanh_F$	`(err-tanh` x`)`	†
$max\_error\_convert_F$	`(err-convert` x`)`	†
$max\_error\_convert_{F'}$	`err-convert-to-string`	†

$max\_error\_convert_{D'}$	`err-convert-to-string`	†

where x and u are expressions of the ISLisp floating point type. Several of the parameter functions are constant.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`(hypotenuse` x y`)`	†
$power_{F,I}(b, z)$	`(expt` b z`)`	⋆
$exp_F(x)$	`(exp` x`)`	⋆
$expm1_F(x)$	`(expm1` x`)`	†
$exp2_F(x)$	`(exp2` x`)`	†
$exp10_F(x)$	`(exp10` x`)`	†
$power_F(b, y)$	`(expt` b y`)` (deviation: `(expt 0 0)` is 1)	⋆
$power1pm1_F(b, y)$	`(exp1pm1` b y`)`	†
$ln_F(x)$	`(log` x`)`	⋆
$ln1p_F(x)$	`(log1p` x`)`	†
$log2_F(x)$	`(log2` x`)`	†
$log10_F(x)$	`(log10` x`)`	†
$logbase_F(b, x)$	`(logbase` b x`)`	†
$logbase1p1p_F(b, x)$	`(logbase1p1p` b x`)`	†
$rad_F(x)$	`(radians` x`)`	†
$axis\_rad_F(x)$	`(axis-rad` x`)`	†
$sin_F(x)$	`(sin` x`)`	⋆
$cos_F(x)$	`(cos` x`)`	⋆
$tan_F(x)$	`(tan` x`)`	⋆
$cot_F(x)$	`(cot` x`)`	†
$sec_F(x)$	`(sec` x`)`	†
$csc_F(x)$	`(csc` x`)`	†
$cossin_F(x)$	`(cossin` x`)`	†
$arcsin_F(x)$	`(asin` x`)`	⋆
$arccos_F(x)$	`(acos` x`)`	⋆
$arctan_F(x)$	`(atan` x`)`	⋆
$arccot_F(x)$	`(acot` x`)`	†
$arccotc_F(x)$	`(acotc` x`)`	†
$arcsec_F(x)$	`(asec` x`)`	†
$arccsc_F(x)$	`(acsc` x`)`	†
$arc_F(x, y)$	`(atan2` y x`)`	⋆
$cycle_F(u, x)$	`(cycle` u x`)`	†
$axis\_cycle_F(u, x)$	`(axis-cycle` u x`)`	†
$sinu_F(u, x)$	`(sinU` u x`)`	†
$cosu_F(u, x)$	`(cosU` u x`)`	†

$tanu_F(u, x)$	`(tanU` u x`)`	†
$cotu_F(u, x)$	`(cotU` u x`)`	†
$secu_F(u, x)$	`(secU` u x`)`	†
$cscu_F(u, x)$	`(cscU` u x`)`	†
$cossinu_F(u, x)$	`(cossinU` u x`)`	†
$arcsinu_F(u, x)$	`(asinU` u x`)`	†
$arccosu_F(u, x)$	`(acosU` u x`)`	†
$arctanu_F(u, x)$	`(atanU` u x`)`	†
$arccotu_F(u, x)$	`(acotU` u x`)`	†
$arccotcu_F(u, x)$	`(acotcU` u x`)`	†
$arcsecu_F(u, x)$	`(asecU` u x`)`	†
$arccscu_F(u, x)$	`(acscU` u x`)`	†
$arcu_F(u, x, y)$	`(atan2U` u y x`)`	†
$rad\_to\_cycle_F(x, w)$	`(rad-to-cycle` x w`)`	†
$cycle\_to\_rad_F(u, x)$	`(cycle-to-rad` u x`)`	†
$cycle\_to\_cycle_F(u, x, w)$	`(cycle-to-cycle` u x w`)`	†
$sinh_F(x)$	`(sinh` x`)`	⋆
$cosh_F(x)$	`(cosh` x`)`	⋆
$tanh_F(x)$	`(tanh` x`)`	⋆
$coth_F(x)$	`(coth` x`)`	†
$sech_F(x)$	`(sech` x`)`	†
$csch_F(x)$	`(csch` x`)`	†
$arcsinh_F(x)$	`(asinh` x`)`	†
$arccosh_F(x)$	`(acosh` x`)`	†
$arctanh_F(x)$	`(atanh` x`)`	⋆
$arccoth_F(x)$	`(acoth` x`)`	†
$arcsech_F(x)$	`(asech` x`)`	†
$arccsch_F(x)$	`(acsch` x`)`	†

where b, x, y, u, and w are expressions of the ISLisp floating point type, and z is an expression of the ISLisp unbounded integer type.

Arithmetic value conversions in ISLisp can be explicit or implicit. The rules for when implicit conversions are done is implementation defined.

$convert_{I \to I''}(x)$	`(format` g `"~B"` x`)`	⋆(binary)
$convert_{I \to I''}(x)$	`(format` g `"~O"` x`)`	⋆(octal)
$convert_{I \to I''}(x)$	`(format` g `"~D"` x`)`	⋆(decimal)
$convert_{I \to I''}(x)$	`(format` g `"~X"` x`)`	⋆(hexadecimal)
$convert_{I \to I''}(x)$	`(format` g `"~`r`R"` x`)`	⋆(radix r)
$convert_{I \to I''}(x)$	`(format-integer` g x r`)`	⋆(radix r)
$floor_{F \to I}(y)$	`(floor` y`)`	⋆
$rounding_{F \to I}(y)$	`(round` y`)`	⋆
$ceiling_{F \to I}(y)$	`(ceiling` y`)`	⋆

$convert_{I \to F}(x)$	`(float` x $kind$`)`	⋆
$convert_{F \to F'}(y)$	`(float` y $kind2$`)`	⋆
$convert_{F \to F''}(y)$	`(format` g `"~G"` y`)`	⋆
$convert_{F \to F''}(y)$	`(format-float` g y`)`	⋆

where x is an expression of the integer type, y is an expression of the floating point type. Conversion from string to numeric value is in ISLisp done via a general read procedure, which reads ISLisp 'S-expressions'.

ISLisp provides non-negative numerals for its integer and floating point datatypes in base 10.

ISLisp does not specify numerals for infinities and NaNs. Suggestion:

$+\infty$	`infinity`	†
qNaN	`nan`	†
sNaN	`signan`	†

as well as string formats for reading and writing these values as character strings.

ISLisp has a notion of 'error' that implies a catchable, possibly returnable, change of control flow. ISLisp uses its exception mechanism as its default means of notification. **underflow** does not cause any notification in ISLisp, and the continuation value to the **underflow** is used directly, since an ISLisp exception is inappropriate. ISLisp uses the error `domain-error` for **invalid** and some **infinitary** notifications, the error `arithmetic-error` for **overflow** notifications, and the error `division-by-zero` for other **infinitary** notifications.

An implementation that wishes to follow LIA-2 should provide recording of indicators as an alternative means of handling numeric notifications. Recording of indicators is the LIA-2 preferred means of handling numeric notifications.

C.10 Modula-2

The programming language Modula-2 is defined by ISO/IEC 10514-1:1996, *Information technology – Programming languages – Part 1: Modula-2, Base Language* [25].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

The Modula-2 datatype `Boolean` corresponds to the LIA datatype **Boolean**.

Every implementation of Modula-2 has at least one integer datatype, and at least one floating point datatype. The notations *INT* and *FLT* are used to stand for the names of one of these datatypes (respectively) in what follows.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	`imax(`x`,` y`)`	†
$min_I(x, y)$	`imin(`x`,` y`)`	†
$max\_seq_I(xs)$	`imaxArr(`xs`)`	†
$min\_seq_I(xs)$	`iminArr(`xs`)`	†
$dim_I(x, y)$	`idim(`x`,` y`)`	†

$shift2_I(x, y)$	`shift2(`x`, `y`)`	†
$shift10_I(x, y)$	`shift10(`x`, `y`)`	†
$power_I(x, y)$	`ipower(`x`, `y`)`	†
$sqrt_I(x)$	`isqrt(`x`)`	†
$divides_I(x, y)$	`divides(`x`, `y`)`	†
$even_I(x)$	`not odd(`x`)`	⋆
$odd_I(x)$	`odd(`x`)`	⋆
$quot_I(x, y)$	`quot(`x`, `y`)`	†
$mod_I(x, y)$	x `mod` y	⋆
$ratio_I(x, y)$	`ratio(`x`, `y`)`	†
$residue_I(x, y)$	`residue(`x`, `y`)`	†
$group_I(x, y)$	`group(`x`, `y`)`	†
$pad_I(x, y)$	`pad(`x`, `y`)`	†
$gcd_I(x, y)$	`gcd(`x`, `y`)`	†
$lcm_I(x, y)$	`lcm(`x`, `y`)`	†
$gcd\_seq_I(xs)$	`gcdArr(`xs`)`	†
$lcm\_seq_I(xs)$	`lcmArr(`xs`)`	†
$add\_wrap_I(x, y)$	`addwrap(`x`, `y`)`	†
$add\_ov_I(x, y)$	`addover(`x`, `y`)`	†
$sub\_wrap_I(x, y)$	`subwrap(`x`, `y`)`	†
$sub\_ov_I(x, y)$	`subover(`x`, `y`)`	†
$mul\_wrap_I(x, y)$	`mulwrap(`x`, `y`)`	†
$mul\_ov_I(x, y)$	`mulover(`x`, `y`)`	†

where x and y are expressions of type *INT* and where xs is an expression of type `array [] of` *INT*.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	`max(`x`, `y`)`	†
$min_F(x, y)$	`min(`x`, `y`)`	†
$mmax_F(x, y)$	`mmax(`x`, `y`)`	†
$mmin_F(x, y)$	`mmin(`x`, `y`)`	†
$max\_seq_F(xs)$	`maxarr(`xs`)`	†
$min\_seq_F(xs)$	`minarr(`xs`)`	†
$mmax\_seq_F(xs)$	`mmaxarr(`xs`)`	†
$mmin\_seq_F(xs)$	`mminarr(`xs`)`	†
$dim_F(x, y)$	`dim(`x`, `y`)`	†
$floor_F(x)$	`floor(`x`)`	†
$floor\_rest_F(x)$	x `- floor(`x`)`	†
$rounding_F(x)$	`rounding(`x`)`	†
$rounding\_rest_F(x)$	x `- rounding(`x`)`	†
$ceiling_F(x)$	`ceiling(`x`)`	†
$ceiling\_rest_F(x)$	x `- ceiling(`x`)`	†

$residue_F(x, y)$	`residue(`x`, `y`)`	†
$sqrt_F(x)$	`sqrt(`x`)`	⋆
$rec\_sqrt_F(x)$	`rec_sqrt(`x`)`	†
$mul_{F \to F'}(x, y)$	`prod(`x`, `y`)`	†
$add\_lo_F(x, y)$	`addlow(`x`, `y`)`	†
$sub\_lo_F(x, y)$	`sublow(`x`, `y`)`	†
$mul\_lo_F(x, y)$	`mullow(`x`, `y`)`	†
$div\_rest_F(x, y)$	`divrest(`x`, `y`)`	†
$sqrt\_rest_F(x)$	`sqrtrest(`x`)`	†

where x and y are expressions of type *FLT*, and where xs is an expression of type `array [] of` *FLT*.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`err_hypotenuse(`x`)`	†
$max\_error\_exp_F$	`err_exp(`x`)`	†
$max\_error\_power_F$	`err_power(`x`)`	†
$big\_angle\_r_F$	`big_radian_angle(`x`)`	†
$max\_error\_rad_F$	`err_rad(`x`)`	†
$max\_error\_sin_F$	`err_sin(`x`)`	†
$max\_error\_tan_F$	`err_tan(`x`)`	†
$min\_angular\_unit_F$	`min_angle_unit(`x`)`	†
$big\_angle\_u_F$	`big_angle(`x`)`	†
$max\_error\_sinu_F(u)$	`err_sin_cycle(`u`)`	†
$max\_error\_tanu_F(u)$	`err_tan_cycle(`u`)`	†
$max\_error\_sinh_F$	`err_sinh(`x`)`	†
$max\_error\_tanh_F$	`err_tanh(`x`)`	†
$max\_error\_convert_F$	`err_convert(`x`)`	†
$max\_error\_convert_{F'}$	`err_convert_to_string`	†
$max\_error\_convert_{D'}$	`err_convert_to_string`	†

where x and u are expressions of type *FLT*. Several of the parameter functions are constant for each type (and library), the argument is then used only to differentiate among the floating point types.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`hypotenuse(`x`, `y`)`	†
$power_{F,I}(b, z)$	`powerI(`b`, `z`)`	†
$exp_F(x)$	`exp(`x`)`	⋆
$expm1_F(x)$	`expm1(`x`)`	†
$exp2_F(x)$	`exp2(`x`)`	†

$exp10_F(x)$	`exp10`(x)	†
$power_F(b, y)$	`power`(b, y)	⋆
$power1pm1_F(b, y)$	`power1PM1`(b, y)	†
$ln_F(x)$	`ln`(x)	⋆
$ln1p_F(x)$	`ln1P`(x)	†
$log2_F(x)$	`log2`(x)	†
$log10_F(x)$	`log10`(x)	†
$logbase_F(b, x)$	`log`(x, b)	†
$logbase1p1p_F(b, x)$	`log1P1P`(x, b)	†
$rad_F(x)$	`radian`(x)	†
$axis\_rad_F(x)$	`axis_rad`(x)	†
$sin_F(x)$	`sin`(x)	⋆
$cos_F(x)$	`cos`(x)	⋆
$tan_F(x)$	`tan`(x)	⋆
$cot_F(x)$	`cot`(x)	†
$sec_F(x)$	`sec`(x)	†
$csc_F(x)$	`csc`(x)	†
$arcsin_F(x)$	`arcsin`(x)	⋆
$arccos_F(x)$	`arccos`(x)	⋆
$arctan_F(x)$	`arctan`(x)	⋆
$arccot_F(x)$	`arccot`(x)	†
$arccotc_F(x)$	`arccotc`(x)	†
$arcsec_F(x)$	`arcsec`(x)	†
$arccsc_F(x)$	`arccsc`(x)	†
$arc_F(x, y)$	`angle`(x, y)	†
$cycle_F(u, x)$	`cycle`(u, x)	†
$axis\_cycle_F(u, x)$	`axis_cycle`(u, x)	†
$sinu_F(u, x)$	`sinu`(u, x)	†
$cosu_F(u, x)$	`cosu`(u, x)	†
$tanu_F(u, x)$	`tanu`(u, x)	†
$cotu_F(u, x)$	`cotu`(u, x)	†
$secu_F(u, x)$	`secu`(u, x)	†
$cscu_F(u, x)$	`cscu`(u, x)	†
$arcsinu_F(u, x)$	`arcsinu`(u, x)	†
$arccosu_F(u, x)$	`arccosu`(u, x)	†
$arctanu_F(u, x)$	`arctanu`(u, x)	†
$arccotu_F(u, x)$	`arccotu`(u, x)	†
$arccotcu_F(u, x)$	`arccotcu`(u, x)	†
$arcsecu_F(u, x)$	`arcsecu`(u, x)	†
$arccscu_F(u, x)$	`arccscu`(u, x)	†
$arcu_F(u, x, y)$	`angleu`(u, x, y)	†

$rad\_to\_cycle_F(x, w)$	`Radian_to_cycle(`x`, `w`)`	†
$cycle\_to\_rad_F(u, x)$	`Cycle_to_radian(`u`, `x`)`	†
$cycle\_to\_cycle_F(u, x, w)$	`Cycle_to_cycle(`u`, `x`, `w`)`	†
$sinh_F(x)$	`sinh(`x`)`	†
$cosh_F(x)$	`cosh(`x`)`	†
$tanh_F(x)$	`tanh(`x`)`	†
$coth_F(x)$	`coth(`x`)`	†
$sech_F(x)$	`sech(`x`)`	†
$csch_F(x)$	`csch(`x`)`	†
$arcsinh_F(x)$	`arcsinh(`x`)`	†
$arccosh_F(x)$	`arccosh(`x`)`	†
$arctanh_F(x)$	`arctanh(`x`)`	†
$arccoth_F(x)$	`arccoth(`x`)`	†
$arcsech_F(x)$	`arcsech(`x`)`	†
$arccsch_F(x)$	`arccsch(`x`)`	†

where b, x, y, u, and w are expressions of type *FLT*, and z is an expression of type *INT*.

Arithmetic value conversions in Modula-2 can be explicit or implicit. The rules for when implicit conversions are applied is not repeated here. The explicit arithmetic value conversions are usually expressed as 'casts', except when converting to/from string formats.

$convert_{I \to I'}(x)$	*INT2*`(`x`)`	⋆
$convert_{I'' \to I'}(f)$	`ReadCard(`f`, `r`)`	⋆
$convert_{I'' \to I}(f)$	`ReadInt(`f`, `r`)`	⋆
$convert_{I' \to I''}(x)$	`WriteCard(`h`, `x`)`	⋆
$convert_{I \to I''}(x)$	`WriteInt(`h`, `x`)`	⋆
$floor_{F \to I}(y)$	`floor(`y`)`	⋆
$rounding_{F \to I}(y)$	`round(`y`)`	⋆
$ceiling_{F \to I}(y)$	`ceiling(`y`)`	⋆
$convert_{I \to F}(x)$	*FLT*`(`x`)`	⋆
$convert_{F \to F'}(y)$	*FLT2*`(`y`)`	⋆
$convert_{F'' \to F}(f)$	`ReadReal(`f`, `z`)`	⋆
$convert_{F \to F''}(y)$	`WriteFloat(`f`, `y`, `a`, `w`)`	⋆
$convert_{F \to F''}(y)$	`WriteEng(`h`, `y`, `a`, `w`)`	⋆
$convert_{F \to F''}(y)$	`WriteReal(`h`, `y`, `a`, `w`)`	⋆
$convert_{D' \to F}(f)$	`ReadReal(`f`, `z`)`	⋆
$convert_{F \to D'}(y)$	`WriteFixed(`h`, `y`, `a`, `w`)`	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*, f is an input file, h is an output file, and a and w are integer expressions. r is an *INT* variable, and z is an *FLT* variable. *INT2* is the integer datatype that corresponds to I'. *FLT2* is the floating point datatype that corresponds to F'.

Modula-2 provides base 8, 10, and 16 non-negative numerals for all its integer types, and base 10 non-negative numerals for all its floating point types. Numerals for floating point types must have a '.' in them. The details are not repeated in this example binding, see ISO/IEC 10514-1, clause 6.8.7.1 Whole Number Literals, and clause 6.8.7.2 Real Literals.

Modula-2 does not specify numerals for infinities and NaNs. Suggestion:

$+\infty$	`INFINITY`	†
qNaN	`NAN`	†
sNaN	`NANSIGNALLING`	†

as well as string formats for reading and writing these values as character strings.

Modula-2 has a notion of 'exception' that implies a non-returnable, but catchable, change of control flow. Modula-2 uses its exception mechanism as its default means of notification. **underflow** does not cause any notification in Modula-2, and the continuation value to the **underflow** is used directly, since a Modula-2 exception is inappropriate for an **underflow** notification. Modula-2 uses the exceptions

a) `REAL-ZERO-DIVISION`,

b) `WHOLE-ZERO-DIVISION`,

c) `WHOLE-ZERO-REMAINDER`,

d) `NEGATIVE-SQRT-ARG`,

e) `NONPOSITIVE-LN-ARG`,

f) `NONPOSITIVE-POWER-ARG`,

g) `ARCSIN-ARG-MAGNITUDE`, and

h) `ARCCOS-ARG-MAGNITUDE`

for **infinitary** and **invalid** notifications. The exceptions `WHOLE-OVERFLOW`, `REAL-OVERFLOW`, and `TAN-OVERFLOW` are used for **overflow** notifications. Since Modula-2 exceptions are non-returnable changes of control flow, no continuation value is provided for these notifications.

An implementation that wishes to follow LIA-2 should provide recording of indicators as an alternative means of handling numeric notifications. Recording of indicators is the LIA-2 preferred means of handling numeric notifications.

C.11 Pascal and Extended Pascal

The programming language Pascal is defined by ISO/IEC 7185:1990, *Information technology - Programming languages – Pascal* [27]. The programming language Extended Pascal is defined by ISO/IEC 10206:1991 *Information technology – Programming languages – Extended Pascal* [28].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

The Pascal datatype `Boolean` corresponds to the LIA datatype **Boolean**.

Every implementation of Pascal has one integer datatype, and one floating point datatype. The notations *INT* and *FLT* are used to stand for the names of one of these datatypes (respectively) in what follows.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	`Imax(`x`, `y`)`	†
$min_I(x, y)$	`Imin(`x`, `y`)`	†
$max\_seq_I(xs)$	`ImaxArr(`xs`)`	†
$min\_seq_I(xs)$	`IminArr(`xs`)`	†
$dim_I(x, y)$	`Idim(`x`, `y`)`	†
$power_I(x, y)$	x `pow` y	⋆(Extended Pascal)
$shift2_I(x, y)$	`shift2(`x`, `y`)`	†
$shift10_I(x, y)$	`shift10(`x`, `y`)`	†
$sqrt_I(x)$	`Isqrt(`x`)`	†
$divides_I(x, y)$	`Divides(`x`, `y`)`	†
$even_I(x)$	`(not Odd(`x`))`	⋆
$odd_I(x)$	`Odd(`x`)`	⋆
$quot_I(x, y)$	`Quotient(`x`, `y`)`	†
$mod_I(x, y)$	`Modulo(`x`, `y`)`	†
$ratio_I(x, y)$	`Ratio(`x`, `y`)`	†
$residue_I(x, y)$	`Residuei(`x`, `y`)`	†
$group_I(x, y)$	`Group(`x`, `y`)`	†
$pad_I(x, y)$	`Pad(`x`, `y`)`	†
$gcd_I(x, y)$	`Gcd(`x`, `y`)`	†
$lcm_I(x, y)$	`Lcm(`x`, `y`)`	†
$gcd\_seq_I(xs)$	`GcdArr(`xs`)`	†
$lcm\_seq_I(xs)$	`LcmArr(`xs`)`	†
$add\_wrap_I(x, y)$	`AddWrap(`x`, `y`)`	†
$add\_ov_I(x, y)$	`AddOver(`x`, `y`)`	†
$sub\_wrap_I(x, y)$	`SubWrap(`x`, `y`)`	†
$sub\_ov_I(x, y)$	`SubOver(`x`, `y`)`	†
$mul\_wrap_I(x, y)$	`MulWrap(`x`, `y`)`	†
$mul\_ov_I(x, y)$	`MulOver(`x`, `y`)`	†

where x and y are expressions of type *INT* and where xs is an expression of type `array [] of` *INT*.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	`Max(`x`, `y`)`	†
$min_F(x, y)$	`Min(`x`, `y`)`	†
$mmax_F(x, y)$	`MMax(`x`, `y`)`	†
$mmin_F(x, y)$	`MMin(`x`, `y`)`	†
$max\_seq_F(xs)$	`MaxArr(`xs`)`	†

$min\_seq_F(xs)$	`MinArr(`xs`)`	†
$mmax\_seq_F(xs)$	`MMaxarr(`xs`)`	†
$mmin\_seq_F(xs)$	`MMinarr(`xs`)`	†
$dim_F(x, y)$	`Dim(`x`, `y`)`	†
$floor_F(x)$	`Floor(`x`)`	†
$floor\_rest_F(x)$	x `- Floor(`x`)`	†
$rounding_F(x)$	`Rounding(`x`)`	†
$rounding\_rest_F(x)$	x `- Rounding(`x`)`	†
$ceiling_F(x)$	`Ceiling(`x`)`	†
$ceiling\_rest_F(x)$	x `- Ceiling(`x`)`	†
$residue_F(x, y)$	`Residue(`x`, `y`)`	†
$sqrt_F(x)$	`Sqrt(`x`)`	⋆
$rec\_sqrt_F(x)$	`Rec_sqrt(`x`)`	†
$mul_{F \to F'}(x, y)$	`Prod(`x`, `y`)`	†
$add\_lo_F(x, y)$	`AddLow(`x`, `y`)`	†
$sub\_lo_F(x, y)$	`SubLow(`x`, `y`)`	†
$mul\_lo_F(x, y)$	`MulLow(`x`, `y`)`	†
$div\_rest_F(x, y)$	`DivRest(`x`, `y`)`	†
$sqrt\_rest_F(x)$	`SqrtRest(`x`)`	†

where x and y are expressions of type *FLT*, and where xs is an expression of type `array [] of` *FLT*.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`Err_hypotenuse(`x`)`	†
$max\_error\_exp_F$	`Err_exp(`x`)`	†
$max\_error\_power_F$	`Err_power(`x`)`	†
$big\_angle\_r_F$	`Big_radian_angle(`x`)`	†
$max\_error\_rad_F$	`Err_rad(`x`)`	†
$max\_error\_sin_F$	`Err_sin(`x`)`	†
$max\_error\_tan_F$	`Err_tan(`x`)`	†
$min\_angular\_unit_F$	`Min_angle_unit(`x`)`	†
$big\_angle\_u_F$	`Big_angle(`x`)`	†
$max\_error\_sinu_F(u)$	`Err_sin_cycle(`u`)`	†
$max\_error\_tanu_F(u)$	`Err_tan_cycle(`u`)`	†
$max\_error\_sinh_F$	`Err_sinh(`x`)`	†
$max\_error\_tanh_F$	`Err_tanh(`x`)`	†
$max\_error\_convert_F$	`Err_convert(`x`)`	†
$max\_error\_convert_{F'}$	`Err_convert_to_string`	†
$max\_error\_convert_{D'}$	`Err_convert_to_string`	†

where x and u are expressions of type *FLT*. Several of the parameter functions are constant for each type (and library), the argument is then used only to differentiate among the floating point types.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`Hypotenuse(`x`, `y`)`	†
$power_{F,I}(b, z)$	b `pow` z	⋆(Extended Pascal)
$exp_F(x)$	`Exp(`x`)`	⋆
$expm1_F(x)$	`ExpM1(`x`)`	†
$exp2_F(x)$	`Exp2(`x`)`	†
$exp10_F(x)$	`Exp10(`x`)`	†
$power_F(b, y)$	b `**` y	⋆(Extended Pascal)
$power1pm1_F(b, y)$	`Power1PM1(`b`, `y`)`	†
$ln_F(x)$	`Ln(`x`)`	⋆
$ln1p_F(x)$	`Ln1P(`x`)`	†
$log2_F(x)$	`Log2(`x`)`	†
$log10_F(x)$	`Log10(`x`)`	†
$logbase_F(b, x)$	`Log(`x`, `b`)`	†
$logbase1p1p_F(b, x)$	`Log1P1P(`x`, `b`)`	†
$rad_F(x)$	`Radian(`x`)`	†
$axis\_rad_F(x)$	`Axis_Radian(`x`, `h`, `v`)`	†
$sin_F(x)$	`Sin(`x`)`	⋆
$cos_F(x)$	`Cos(`x`)`	⋆
$tan_F(x)$	`Tan(`x`)`	†
$cot_F(x)$	`Cot(`x`)`	†
$sec_F(x)$	`Sec(`x`)`	†
$csc_F(x)$	`Csc(`x`)`	†
$arcsin_F(x)$	`Arcsin(`x`)`	†
$arccos_F(x)$	`Arccos(`x`)`	†
$arctan_F(x)$	`Arctan(`x`)`	⋆
$arccot_F(x)$	`Arccot(`x`)`	†
$arccotc_F(x)$	`Arccotc(`x`)`	†
$arcsec_F(x)$	`Arcsec(`x`)`	†
$arccsc_F(x)$	`Arccsc(`x`)`	†
$arc_F(x, y)$	`Angle(`x`, `y`)`	†
$cycle_F(u, x)$	`Cycle(`u`, `x`)`	†
$axis\_cycle_F(u, x)$	`Axis_Cycle(`u`, `x`, `h`, `v`)`	†
$sinu_F(u, x)$	`SinU(`u`, `x`)`	†
$cosu_F(u, x)$	`CosU(`u`, `x`)`	†
$tanu_F(u, x)$	`TanU(`u`, `x`)`	†
$cotu_F(u, x)$	`CotU(`u`, `x`)`	†

$secu_F(u, x)$	`SecU(`u`, `x`)`	†
$cscu_F(u, x)$	`CscU(`u`, `x`)`	†
$arcsinu_F(u, x)$	`ArcsinU(`u`, `x`)`	†
$arccosu_F(u, x)$	`ArccosU(`u`, `x`)`	†
$arctanu_F(u, x)$	`ArctanU(`u`, `x`)`	†
$arccotu_F(u, x)$	`ArccotU(`u`, `x`)`	†
$arccotcu_F(u, x)$	`ArccotcU(`u`, `x`)`	†
$arcsecu_F(u, x)$	`ArcsecU(`u`, `x`)`	†
$arccscu_F(u, x)$	`ArccscU(`u`, `x`)`	†
$arcu_F(u, x, y)$	`AngleU(`u`, `x`, `y`)`	†
$rad\_to\_cycle_F(x, w)$	`RadianToCycle(`x`, `w`)`	†
$cycle\_to\_rad_F(u, x)$	`CycleToRadian(`u`, `x`)`	†
$cycle\_to\_cycle_F(u, x, w)$	`CycleToCycle(`u`, `x`, `w`)`	†
$sinh_F(x)$	`Sinh(`x`)`	†
$cosh_F(x)$	`Cosh(`x`)`	†
$tanh_F(x)$	`Tanh(`x`)`	†
$coth_F(x)$	`Coth(`x`)`	†
$sech_F(x)$	`Sech(`x`)`	†
$csch_F(x)$	`Csch(`x`)`	†
$arcsinh_F(x)$	`Arcsinh(`x`)`	†
$arccosh_F(x)$	`Arccosh(`x`)`	†
$arctanh_F(x)$	`Arctanh(`x`)`	†
$arccoth_F(x)$	`Arccoth(`x`)`	†
$arcsech_F(x)$	`Arcsech(`x`)`	†
$arccsch_F(x)$	`Arccsch(`x`)`	†

where b, x, y, u, and w are expressions of type *FLT*, h and v are variables of type *FLT*, and z is an expression of type *INT*.

Arithmetic value conversions in Pascal can be explicit or implicit. The rules for when implicit conversions are applied are not repeated here.

$convert_{I'' \to I}(f)$	`read(`f`?, `r`)`	⋆
$convert_{I \to I''}(x)$	`write(`h`?, `x`:`n`?)`	⋆
$floor_{F \to I}(y)$	`floor(`y`)`	†
$rounding_{F \to I}(y)$	`round(`y`)`	⋆
$ceiling_{F \to I}(y)$	`ceiling(`y`)`	†
$convert_{F'' \to F}(f)$	`read(`f`?, `m`)`	⋆
$convert_{F \to F''}(y)$	`write(`h`?, `y`:`i`)`	⋆
$convert_{D' \to F}(f)$	`read(`f`?, `m`)`	⋆
$convert_{F \to D'}(y)$	`write(`h`?, `y`:`i`:`a`)`	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*, f is an input file, h is an output file, r is an integer variable, m is a floating point variable, n, i and a are integer literals. A ? above indicates that the parameter is optional.

Pascal provides base 10 non-negative numerals for its only integer type and only floating point type. Numerals for floating point types must have a '.' in them. The details are not repeated in this example binding, see ISO/IEC 7185:1990 and ISO/IEC 10206:1991.

Pascal does not specify numerals for infinities and NaNs. Suggestion:

$+\infty$	`INFINITY`	†
qNaN	`NAN`	†
sNaN	`NANSIGNALLING`	†

as well as string formats for reading and writing these values as character strings.

Pascal has a notion of 'error', which results in a change of 'control flow', which cannot be returned from, nor caught. An 'error' results in the termination of the program. **infinitary** for integer types and **invalid** (in general) are considered to be 'errors'. No notification results for **underflow**, and the continuation value (specified by LIA-2) is used directly, since recording of indicators is not available and 'error' is inappropriate for **underflow**. The handling of integer **overflow** is implementation dependent. The handling of floating point **overflow** and **infinitary** should be to return a suitable infinity (specified by LIA-2), if possible, without any notification, since recording of indicators is not available.

An implementation that wishes to follow LIA-2 should provide recording of indicators as an alternative means of handling numeric notifications. Recording of indicators is the LIA-2 preferred means of handling numeric notifications.

C.12 PL/I

The programming language PL/I is defined by ANSI X3.53-1976 (R1998), *Programming languages – PL/I* [43], and endorsed by ISO 6160:1979, *Programming languages – PL/I* [29]. The programming language General Purpose PL/I is defined by ISO/IEC 6522:1992, *Information technology – Programming languages – PL/I general-purpose subset* [30], also: ANSI X3.74-1987 (R1998).

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

The LIA datatype **Boolean** is implemented in the PL/I datatype `BIT(1)` (1 = **true** and 0 = **false**).

An implementation of PL/I provides at least one integer datatype, and at least one floating point datatype. The attribute `FIXED(`n`,0)` selects a signed integer datatype with at least n (decimal or binary) digits of storage. The attribute `FLOAT(`k`)` selects a floating point datatype with at least n (decimal or binary) digits of precision. The notation *INT* is used to stand for the name of one of the integer datatypes, and *FLT* is used to stand for the name of one of the floating point datatypes in what follows.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	`max(`x`, `y`)`	⋆
$min_I(x, y)$	`min(`x`, `y`)`	⋆
$max\_seq_I(xs)$	`max(`x_1`, `x_2`, ..., `x_n`)`	⋆
$min\_seq_I(xs)$	`min(`x_1`, `x_2`, ..., `x_n`)`	⋆
$dim_I(x, y)$	`dim(`x`, `y`)`	†
$power_I(x, y)$	x `**` y	⋆
$shift2_I(x, y)$	`shift2(`x`, `y`)`	†
$shift10_I(x, y)$	`shift10(`x`, `y`)`	†
$sqrt_I(x)$	`sqrt(`x`)`	†
$divides_I(x, y)$	`divides(`x`, `y`)`	†
$even_I(x)$	`mod(`x`, 2) = 0`	⋆
$odd_I(x)$	`mod(`x`, 2) ¬= 0`	⋆
$quot_I(x, y)$	`quotient(`x`, `y`)`	†
$mod_I(x, y)$	`mod(`x`, `y`)`	⋆
$ratio_I(x, y)$	`ratio(`x`, `y`)`	†
$residue_I(x, y)$	`residue(`x`, `y`)`	†
$group_I(x, y)$	`group(`x`, `y`)`	†
$pad_I(x, y)$	`pad(`x`, `y`)`	†
$gcd_I(x, y)$	`gcd(`x`, `y`)`	†
$lcm_I(x, y)$	`lcm(`x`, `y`)`	†
$gcd\_seq_I(xs)$	`gcd(`x_1`, `x_2`, ..., `x_n`)`	†
$lcm\_seq_I(xs)$	`lcm(`x_1`, `x_2`, ..., `x_n`)`	†
$add\_wrap_I(x, y)$	`add_wrap(`x`, `y`)`	†
$add\_ov_I(x, y)$	`add_over(`x`, `y`)`	†
$sub\_wrap_I(x, y)$	`sub_wrap(`x`, `y`)`	†
$sub\_ov_I(x, y)$	`sub_over(`x`, `y`)`	†
$mul\_wrap_I(x, y)$	`mul_wrap(`x`, `y`)`	†
$mul\_ov_I(x, y)$	`mul_over(`x`, `y`)`	†

where x and y are expressions of type *INT*, and where are x_1, x_2, ..., x_n expressions of type array of *INT*.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	`max(`x`, `y`)`	⋆
$min_F(x, y)$	`min(`x`, `y`)`	⋆
$mmax_F(x, y)$	`mmax(`x`, `y`)`	†
$mmin_F(x, y)$	`mmin(`x`, `y`)`	†
$max\_seq_F(xs)$	`max(`x_1`, `x_2`, ..., `x_n`)`	⋆
$min\_seq_F(xs)$	`min(`x_1`, `x_2`, ..., `x_n`)`	⋆
$mmax\_seq_F(xs)$	`mmax(`x_1`, `x_2`, ..., `x_n`)`	†
$mmin\_seq_F(xs)$	`mmin(`x_1`, `x_2`, ..., `x_n`)`	†
$dim_F(x, y)$	`dim(`x`, `y`)`	†

$floor_F(x)$	`floor(`x`)`	⋆
$floor\_rest_F(x)$	x `- floor(`x`)`	⋆
$rounding_F(x)$	`round(`x`)`	†
$rounding\_rest_F(x)$	x `- round(`x`)`	†
$ceiling_F(x)$	`ceil(`x`)`	⋆
$ceiling\_rest_F(x)$	x `- ceil(`x`)`	⋆
$residue_F(x, y)$	`residue(`x`, `y`)`	†
$sqrt_F(x)$	`sqrt(`x`)`	⋆
$rec\_sqrt_F(x)$	`rec_sqrt(`x`)`	†
$mul_{F \rightarrow F'}(x, y)$	`prod(`x`, `y`)`	†
$add\_lo_F(x, y)$	`add_low(`x`, `y`)`	†
$sub\_lo_F(x, y)$	`sub_low(`x`, `y`)`	†
$mul\_lo_F(x, y)$	`mul_low(`x`, `y`)`	†
$div\_rest_F(x, y)$	`div_rest(`x`, `y`)`	†
$sqrt\_rest_F(x)$	`sqrt_rest(`x`)`	†

where x and y are expressions of type *FLT*, and where xs is an expression of type array of *FLT*.

The parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`err_hypotenuse(`x`)`	†
$max\_error\_exp_F$	`err_exp(`x`)`	†
$max\_error\_power_F$	`err_power(`x`)`	†
$big\_angle\_r_F$	`big_radian_angle(`x`)`	†
$max\_error\_rad_F$	`err_rad(`x`)`	†
$max\_error\_sin_F$	`err_sin(`x`)`	†
$max\_error\_tan_F$	`err_tan(`x`)`	†
$min\_angular\_unit_F$	`min_angle_unit(`x`)`	†
$big\_angle\_u_F$	`big_angle(`x`)`	†
$max\_error\_sinu_F(u)$	`err_sin_cycle(`u`)`	†
$max\_error\_tanu_F(u)$	`err_tan_cycle(`u`)`	†
$max\_error\_sinh_F$	`err_sinh(`x`)`	†
$max\_error\_tanh_F$	`err_tanh(`x`)`	†
$max\_error\_convert_F$	`err_convert(`x`)`	†
$max\_error\_convert_{F'}$	`err_convert_to_string`	†
$max\_error\_convert_{D'}$	`err_convert_to_string`	†

where x and u are expressions of type *FLT*. Several of the parameter functions are constant for each type (and library), the argument is then used only to differentiate among the floating point types.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`hypotenuse(`x`, `y`)`	†

$power_{F,I}(b, z)$	`poweri(`b`, `z`)`	†
$exp_F(x)$	`exp(`x`)`	⋆
$expm1_F(x)$	`expm1(`x`)`	†
$exp2_F(x)$	`exp2(`x`)`	†
$exp10_F(x)$	`exp10(`x`)`	†
$power_F(b, y)$	`power(`b`, `y`)`	†
$power1pm1_F(b, y)$	`power1pm1(`b`, `y`)`	†
$ln_F(x)$	`log(`x`)`	⋆
$ln1p_F(x)$	`log1p(`x`)`	†
$log2_F(x)$	`log2(`x`)`	⋆
$log10_F(x)$	`log10(`x`)`	⋆
$logbase_F(b, x)$	`log(`b`, `x`)`	†
$logbase1p1p_F(b, x)$	`log1p1p(`b`, `x`)`	†
$rad_F(x)$	`rad(`x`)`	†
$axis\_rad_F(x)$	`axis_rad(`x`)`	†
$sin_F(x)$	`sin(`x`)`	⋆
$cos_F(x)$	`cos(`x`)`	⋆
$tan_F(x)$	`tan(`x`)`	⋆
$cot_F(x)$	`cot(`x`)`	⋆
$sec_F(x)$	`sec(`x`)`	†
$csc_F(x)$	`csc(`x`)`	†
$arcsin_F(x)$	`arcsin(`x`)`	⋆
$arccos_F(x)$	`arccos(`x`)`	⋆
$arctan_F(x)$	`arctan(`x`)`	⋆
$arccot_F(x)$	`arccot(`x`)`	†
$arccotc_F(x)$	`arccotc(`x`)`	†
$arcsec_F(x)$	`arcsec(`x`)`	†
$arccsc_F(x)$	`arccsc(`x`)`	†
$arc_F(x, y)$	`arc(`x`, `y`)`	†
$cycle_F(u, x)$	`cycle(`u`, `x`)`	†
$axis\_cycle_F(u, x)$	`axis_cycle(`u`, `x`)`	†
$sinu_F(u, x)$	`sin(`u`, `x`)`	†
$cosu_F(u, x)$	`cos(`u`, `x`)`	†
$tanu_F(u, x)$	`tan(`u`, `x`)`	†
$cotu_F(u, x)$	`cot(`u`, `x`)`	†
$secu_F(u, x)$	`sec(`u`, `x`)`	†
$cscu_F(u, x)$	`csc(`u`, `x`)`	†
$arcsinu_F(u, x)$	`arcsin(`u`, `x`)`	†
$arccosu_F(u, x)$	`arccos(`u`, `x`)`	†
$arctanu_F(u, x)$	`arctan(`u`, `x`)`	†

$arccotu_F(u, x)$	`arccot(`u`, `x`)`	†
$arccotcu_F(u, x)$	`arccotc(`u`, `x`)`	†
$arcsecu_F(u, x)$	`arcsec(`u`, `x`)`	†
$arccscu_F(u, x)$	`arccsc(`u`, `x`)`	†
$arcu_F(u, x, y)$	`arc(`u`, `x`, `y`)`	†
$sinu_F(360, x)$	`sind(`x`)`	⋆
$cosu_F(360, x)$	`cosd(`x`)`	⋆
$tanu_F(360, x)$	`tand(`x`)`	⋆
$cotu_F(360, x)$	`cotd(`x`)`	⋆
$secu_F(360, x)$	`secd(`x`)`	†
$cscu_F(360, x)$	`cscd(`x`)`	†
$arcsinu_F(360, x)$	`arcsind(`x`)`	⋆
$arccosu_F(360, x)$	`arccosd(`x`)`	⋆
$arctanu_F(360, x)$	`arctand(`x`)`	⋆
$arccotu_F(360, x)$	`arccotd(`x`)`	†
$arccotcu_F(360, x)$	`arccotcd(`x`)`	†
$arcsecu_F(360, x)$	`arcsecd(`x`)`	†
$arccscu_F(360, x)$	`arccscd(`x`)`	†
$arcu_F(360, x, y)$	`arcd(`x`, `y`)`	†
$rad\_to\_cycle_F(x, w)$	`rad_to_cycle(`x`, `w`)`	†
$cycle\_to\_rad_F(u, x)$	`cycle_to_rad(`u`, `x`)`	†
$cycle\_to\_cycle_F(u, x, w)$	`cycle_to_cycle(`u`, `x`, `w`)`	†
$sinh_F(x)$	`sinh(`x`)`	⋆
$cosh_F(x)$	`cosh(`x`)`	⋆
$tanh_F(x)$	`tanh(`x`)`	⋆
$coth_F(x)$	`coth(`x`)`	†
$sech_F(x)$	`sech(`x`)`	†
$csch_F(x)$	`csch(`x`)`	†
$arcsinh_F(x)$	`arcsinh(`x`)`	⋆
$arccosh_F(x)$	`arccosh(`x`)`	⋆
$arctanh_F(x)$	`arctanh(`x`)`	⋆
$arccoth_F(x)$	`arccoth(`x`)`	†
$arcsech_F(x)$	`arcsech(`x`)`	†
$arccsch_F(x)$	`arccsch(`x`)`	†

where b, x, y, u, and w are expressions of type *FLT*, and z is an expression of type *INT*.

Arithmetic value conversions in PL/I can be explicit or implicit. The rules for when implicit conversions are applied is not repeated here.

$convert_{I \to I'}(x)$	`FIXED(`x`, `p`)`	⋆
$convert_{I'' \to I}(f)$	`GET FILE (`f`)?  EDIT (`r`) (F(`w`));`	⋆
$convert_{I \to I''}(x)$	`PUT FILE (`h`)?  EDIT (`x`) (F(`w`));`	⋆
$floor_{F \to I}(y)$	`FIXED(floor(`y`), `p`)`	⋆

$rounding_{F\rightarrow I}(y)$	`FIXED(round(y),p)`	†
$ceiling_{F\rightarrow I}(y)$	`FIXED(ceil(y), p)`	⋆
$convert_{I\rightarrow F}(x)$	`FLOAT(x, p)`	⋆
$convert_{I\rightarrow F}(x)$	`DECIMAL(x, p)`	⋆
$convert_{I\rightarrow F}(x)$	`BINARY(x, p)`	⋆
$convert_{F\rightarrow F'}(y)$	`FLOAT(y, p)`	⋆
$convert_{F\rightarrow F'}(y)$	`DECIMAL(y, p)`	⋆
$convert_{F\rightarrow F'}(y)$	`BINARY(y, p)`	⋆
$convert_{F''\rightarrow F}(f)$	`GET FILE (f)?  EDIT (t) (E(w,a));`	⋆
$convert_{F\rightarrow F''}(y)$	`PUT FILE (h)?  EDIT (y) (E(w,a));`	⋆
$convert_{D'\rightarrow F}(f)$	`GET FILE (f)?  EDIT (t) (F(w,a));`	⋆
$convert_{F\rightarrow D'}(y)$	`FIXED(y, p, a)`	⋆
$convert_{F\rightarrow D'}(y)$	`PUT FILE (h)?  EDIT (y) (F(w,a));`	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*, f is an input file, h is an output file, r is an integer variable, t is a floating point variable, p, w, and a are positive integer values. A ? above indicates that the parameter is optional.

PL/I provides base 10 non-negative numerals for all its integer and floating point types.

PL/I does not specify numerals for infinities and NaNs. Suggestion:

$+\infty$	`INFINITY`	†
qNaN	`NAN`	†
sNaN	`NANSIGNALLING`	†

as well as string formats for reading and writing these values as character strings.

PL/I has a notion of 'condition' that implies a non-returnable, but catchable (in an `ON`-unit), change of control flow. PL/I uses its condition mechanism as its default means of notification. PL/I uses the condition `UNDERFLOW` for **underflow** notifications. PL/I uses the condition `ZERODIVIDE` for **infinitary** notifications, and the conditions `FIXEDOVERFLOW`, `SIZE`, and `OVERFLOW` for **overflow** notifications, and the exception `UNDEFINED` (†) for **invalid** notifications. Since PL/I exceptions are non-returnable changes of control flow, no continuation value is provided for these notifications. This is inappropriate, especially for underflow, so `UNDERFLOW` notifications are ignored if there is no `ON`-unit for `UNDERFLOW` in the program.

An implementation that wishes to follow LIA-2 should provide recording of indicators as an alternative means of handling numeric notifications. Recording of indicators is the LIA-2 preferred means of handling numeric notifications.

C.13 SML

The programming language SML is defined by *The Definition of Standard ML (Revised)* [67].

An implementation should follow all the requirements of LIA-2 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and should be provided by an implementation that wishes to conform to the LIA-2 for that operation. For each of the marked items a suggested identifier is provided.

The SML datatype `Boolean` corresponds to the LIA datatype **Boolean**.

Every implementation of SML has at least one integer datatype, `int`, and at least one floating point datatype, `real`. The notation *INT* is used to stand for the name of one of the integer datatypes, and *FLT* is used to stand for the name of one of the floating point datatypes in what follows.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

$max_I(x, y)$	x `max` y or `op max (`x`,` y`)`	⋆
$min_I(x, y)$	x `min` y or `op min (`x`,` y`)`	⋆
$max\_seq_I(xs)$	`maximum` xs	†
$min\_seq_I(xs)$	`minimum` xs	†
$dim_I(x, y)$	x `dim` y or `op dim (`x`,` y`)`	†
$power_I(x, y)$	x `pow` y or `op pow (`x`,` y`)`	†
$shift2_I(x, y)$	`shift2(`x`,` y`)`	†
$shift10_I(x, y)$	`shift10(`x`,` y`)`	†
$sqrt_I(x)$	`isqrt` x	†
$divides_I(x, y)$	`divides (`x`,` y`)`	†
$even_I(x)$	`even` x	†
$odd_I(x)$	`odd` x	†
$quot_I(x, y)$	x `div` y or `op div (`x`,` y`)`	⋆
$mod_I(x, y)$	x `mod` y or `op mod (`x`,` y`)`	⋆
$ratio_I(x, y)$	`ratio (`x`,` y`)`	†
$residue_I(x, y)$	`residue (`x`,` y`)`	†
$group_I(x, y)$	`group (`x`,` y`)`	†
$pad_I(x, y)$	`pad (`x`,` y`)`	†
$gcd_I(x, y)$	`gcd (`x`,` y`)`	⋆
$lcm_I(x, y)$	`lcm (`x`,` y`)`	⋆
$gcd\_seq_I(xs)$	`gcd_seq` xs	†
$lcm\_seq_I(xs)$	`lcm_seq` xs	†
$add\_wrap_I(x, y)$	x `+:` y	†
$add\_ov_I(x, y)$	x `+:+` y	†
$sub\_wrap_I(x, y)$	x `-:` y	†
$sub\_ov_I(x, y)$	x `-:+` y	†
$mul\_wrap_I(x, y)$	x `*:` y	†
$mul\_ov_I(x, y)$	x `*:+` y	†

where x and y are expressions of type *INT* and where xs is an expression of type *INT* `list`.

The LIA-2 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

$max_F(x, y)$	x `max` y or `op max (`x`,` y`)`	⋆

$min_F(x, y)$	x `min` y or `op min (`x`, `y`)`	⋆
$mmax_F(x, y)$	x `mmax` y or `op mmax (`x`, `y`)`	†
$mmin_F(x, y)$	x `mmin` y or `op mmin (`x`, `y`)`	†
$max\_seq_F(xs)$	`maximum` xs	†
$min\_seq_F(xs)$	`minimum` xs	†
$mmax\_seq_F(xs)$	`mmaximum` xs	†
$mmin\_seq_F(xs)$	`mminimum` xs	†
$dim_F(x, y)$	`dim (`x`, `y`)`	†
$floor_F(x)$	`realFloor` x	⋆
$floor\_rest_F(x)$	x `- realFloor` x	⋆
$rounding_F(x)$	`realRound` x	†
$rounding\_rest_F(x)$	x `- realRound` x	†
$ceiling_F(x)$	`realCeil` x	⋆
$ceiling\_rest_F(x)$	x `- realCeil` x	⋆
$residue_F(x, y)$	`residue (`x`, `y`)`	†
$sqrt_F(x)$	`sqrt` x	⋆
$rec\_sqrt_F(x)$	`rec_sqrt` x	†
$mul_{F \to F'}(x, y)$	`prod (`x`, `y`)`	†
$add\_lo_F(x, y)$	x `+:-` y	†
$sub\_lo_F(x, y)$	x `-:-` y	†
$mul\_lo_F(x, y)$	x `*:-` y	†
$div\_rest_F(x, y)$	x `/:*` y	†
$sqrt\_rest_F(x)$	`sqrt_rest` x	†

where x and y are expressions of type *FLT*, and where xs is an expression of type *FLT* `list`.

The parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

$max\_error\_hypot_F$	`err_hypotenuse` x	†
$max\_error\_exp_F$	`err_exp` x	†
$max\_error\_power_F$	`err_power` x	†
$big\_angle\_r_F$	`big_radian_angle` x	†
$max\_error\_radn_F$	`err_rad` x	†
$max\_error\_sin_F$	`err_sin` x	†
$max\_error\_tan_F$	`err_tan` x	†
$min\_angular\_unit_F$	`min_angular_unit` x	†
$big\_angle\_u_F$	`big_angle` x	†
$max\_error\_sinu_F(u)$	`err_sin_cycle` u	†
$max\_error\_tanu_F(u)$	`err_tan_cycle` u	†
$max\_error\_sinh_F$	`err_sinh` x	†
$max\_error\_tanh_F$	`err_tanh` x	†
$max\_error\_convert_F$	`err_convert` x	†

$max\_error\_convert_{F'}$	`err_convert_to_string`	†
$max\_error\_convert_{D'}$	`err_convert_to_string`	†

where x and u are expressions of type *FLT*. Several of the parameter functions are constant for each type, the argument is then used only to differentiate among the floating point types.

The LIA-2 elementary floating point operations are listed below, along with the syntax used to invoke them:

$hypot_F(x, y)$	`hypotenuse (`x`,`y`)`	†
$power_{F,I}(b, z)$	b `^^` z or `op ^^ (`b`,`z`)`	†
$exp_F(x)$	`exp` x	⋆
$expm1_F(x)$	`expM1` x	†
$exp2_F(x)$	`exp2` x	†
$exp10_F(x)$	`exp10` x	†
$power_F(b, y)$	b `**` y or `op ** (`b`, `y`)`	†
$pow_F(b, y)$	b `pow` y or `op pow (`b`, `y`)`	⋆ Not LIA-2! (See C.)
$power1pm1_F(b, y)$	`power1PM1 (`b`, `y`)`	†
$ln_F(x)$	`ln` x	⋆
$ln1p_F(x)$	`ln1P` x	†
$log2_F(x)$	`log2` x	†
$log10_F(x)$	`log10` x	⋆
$logbase_F(b, x)$	`log_base (`b`, `x`)`	†
$logbase1p1p_F(b, x)$	`log_base1P1P (`b`,`x`)`	†
$rad_F(x)$	`radians` x	†
$axis\_rad_F(x)$	`axis_radians` x	†
$sin_F(x)$	`sin` x	⋆
$cos_F(x)$	`cos` x	⋆
$tan_F(x)$	`tan` x	⋆
$cot_F(x)$	`cot` x	†
$sec_F(x)$	`sec` x	†
$csc_F(x)$	`csc` x	†
$arcsin_F(x)$	`arcsin` x	⋆
$arccos_F(x)$	`arccos` x	⋆
$arctan_F(x)$	`arctan` x	⋆
$arccot_F(x)$	`arccot` x	†
$arccotc_F(x)$	`arccotc` x	†
$arcsec_F(x)$	`arcsec` x	†
$arccsc_F(x)$	`arccsc` x	†
$arc_F(x, y)$	`arctan2 (`y`, `x`)`	⋆
$cycle_F(u, x)$	`cycle` u x	†
$axis\_cycle_F(u, x)$	`axis_cycle` u x	†
$sinu_F(u, x)$	`sinU` u x	†

$cosu_F(u,x)$	`cosU` u x	†
$tanu_F(u,x)$	`tanU` u x	†
$cotu_F(u,x)$	`cotU` u x	†
$secu_F(u,x)$	`secU` u x	†
$cscu_F(u,x)$	`cscU` u x	†
$arcsinu_F(u,x)$	`arcsinU` u x	†
$arccosu_F(u,x)$	`arccosU` u x	†
$arctanu_F(u,x)$	`arctanU` u x	†
$arccotu_F(u,x)$	`arccotU` u x	†
$arccotcu_F(u,x)$	`arccotcU` u x	†
$arcsecu_F(u,x)$	`arcsecU` u x	†
$arccscu_F(u,x)$	`arccscU` u x	†
$arcu_F(u,x,y)$	`arctan2U` u `(`y`,` x`)`	†
$rad\_to\_cycle_F(x,w)$	`rad_to_cycle` w x	†
$cycle\_to\_rad_F(u,x)$	`cycle_to_rad` u x	†
$cycle\_to\_cycle_F(u,x,w)$	`cycle_to_cycle` u w x	†
$sinh_F(x)$	`sinh` x	⋆
$cosh_F(x)$	`cosh` x	⋆
$tanh_F(x)$	`tanh` x	⋆
$coth_F(x)$	`coth` x	†
$sech_F(x)$	`sech` x	†
$csch_F(x)$	`csch` x	†
$arcsinh_F(x)$	`arcsinh` x	†
$arccosh_F(x)$	`arccosh` x	†
$arctanh_F(x)$	`arctanh` x	†
$arccoth_F(x)$	`arccoth` x	†
$arcsech_F(x)$	`arcsech` x	†
$arccsch_F(x)$	`arccsch` x	†

where b, x, y, u, and w are expressions of type *FLT*, and z is an expression of type *INT*.

Type conversions in SML are always explicit.

$convert_{I \to I'}(x)$	`fromLarge` x or `toLarge` x	⋆
$convert_{I'' \to I}(s)$	`fromString` s	⋆
$convert_{I'' \to I}(s)$	`scan` *radix* *getc* s	⋆
$convert_{I \to I''}(x)$	`toString` x	⋆
$floor_{F \to I}(y)$	`floor` y	⋆
$floor_{F \to I}(y)$	`toInt IEEEReal.TO_NEGINF` y	⋆
$floor_{F \to I}(y)$	`toLargeInt IEEEReal.TO_NEGINF` y	⋆
$rounding_{F \to I}(y)$	`round` y	⋆
$rounding_{F \to I}(y)$	`toInt IEEEReal.TO_NEAREST` y	⋆
$rounding_{F \to I}(y)$	`toLargeInt IEEEReal.TO_NEAREST` y	⋆
$ceiling_{F \to I}(y)$	`ceiling` y	⋆
$ceiling_{F \to I}(y)$	`toInt IEEEReal.TO_POSINF` y	⋆

$ceiling_{F \to I}(y)$	`toLargeInt IEEEReal.TO_POSINF` y	⋆
$convert_{I \to F}(x)$	`fromInt` x	⋆
$convert_{I \to F}(x)$	`fromLargeInt` x	⋆
$convert_{F \to F'}(y)$	`toLarge` y	⋆
$convert_{F \to F'}(y)$	`fromLarge IEEEReal.TO_NEAREST` y	⋆
$convert_{F'' \to F}(s)$	`fromString` s	⋆
$convert_{F'' \to F}(s)$	`fromDecimal` s	⋆
$convert_{F'' \to F}(s)$	`scan` $getc$ s	⋆
$convert_{F \to F''}(y)$	`fmt (SCI` a`)` y	⋆
$convert_{F \to F''}(y)$	`toDecimal` y	⋆
$convert_{D' \to F}(s)$	`fromString` s	⋆
$convert_{D' \to F}(s)$	`scan` $getc$ s	⋆
$convert_{F \to D'}(y)$	`fmt (FIX` a`)` y	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*, s is a string, *radix* is an integer ($\geqslant 2$), *getc* is a character reading method, and a is an integer.

SML provides non-negative base 10 numerals for all its integer and floating point types. There is no differentiation between the numerals for different floating point types, nor between numerals for different integer types, but integer numerals cannot be used for floating point values. The details are not repeated in this example binding, see *The Definition of Standard ML (Revised)* [67].

SML specifies numerals for infinities, but not NaNs:

$+\infty$	`posInf`	⋆
$-\infty$	`negInf`	⋆
qNaN	`NaN`	†
sNaN	`sigNaN`	†

An implementation wishing to conform to LIA-2 should also provide string formats for reading and writing these values as character strings.

SML has a notion of 'exception' that implies a non-returnable, but catchable, change of control flow. SML uses its exception mechanism as its default means of notification. **underflow** does not cause any notification in SML, and the continuation value to the **underflow** is used directly, since an SML exception is inappropriate for an **underflow** notification. SML uses the exception `Div` for **infinitary** notifications, the exception `Overflow` for **overflow** notifications, and the exception `Domain` for **invalid** notifications (except for `sin`, `cos`, or `tan` given an infinitary argument, where the **invalid** notification is ignored). Since SML exceptions are non-returnable changes of control flow, no continuation value is provided for these notifications.

An implementation that wishes to follow LIA-2 should provide recording of indicators as an alternative means of handling numeric notifications. Recording of indicators is the LIA-2 preferred means of handling numeric notifications.

Annex D
(informative)

Bibliography

This annex gives references to publications relevant to LIA-2.

International standards documents

[1] ISO/IEC Directives, Part 3: *Rules for the structure and drafting of International Standards*, 1997.

[2] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems.* (Also: ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*)

[3] ISO/IEC 10967-3, *Information technology – Language independent arithmetic – Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions*, (LIA-3). (To be published.)

[4] ISO 6093:1985, *Information processing – Representation of numerical values in character strings for information interchange.*

[5] ISO/IEC 10646-1:2000, *Information technology – Universal multi-octet character set (UCS) – Part 1: Architecture and Basic Multilingual plane*, second edition.

[6] ISO/IEC 10646-2:2001, *Information technology – Universal multi-octet character set (UCS) – Part 2: Supplementary planes.*

[7] ISO/IEC TR 10176:1998, *Information technology – Guidelines for the preparation of programming language standards.*

[8] ISO/IEC TR 10182:1993, *Information technology – Programming languages, their environments and system software interfaces – Guidelines for language bindings.*

[9] ISO/IEC 13886:1996, *Information technology – Language-Independent Procedure Calling*, (LIPC).

[10] ISO/IEC 11404:1996, *Information technology – Programming languages, their environments and system software interfaces – Language-independent datatypes*, (LID).

[11] ISO/IEC 8652:1995, *Information technology – Programming languages – Ada.*

[12] ISO/IEC 13813:1998, *Information technology – Programming languages – Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types).*

[13] ISO/IEC 13814:1998, *Information technology – Programming languages – Generic package of complex elementary functions for Ada.*

[14] ISO 8485:1989, *Programming languages – APL.*

[15] ISO/IEC 13751:2001, *Information technology – Programming languages, their environments and system software interfaces – Programming language extended APL.*

[16] ISO/IEC 10279:1991, *Information technology – Programming languages – Full BASIC.* (Essentially an endorsement of ANSI X3.113-1987 (R1998) [40].)

[17] ISO/IEC 9899:1999, *Programming languages – C.*

[18] ISO/IEC 14882:1998, *Programming languages – C++.*

[19] ISO 1989:1985, *Programming languages – COBOL.* (Endorsement of ANSI X3.23-1985 (R1991) [41].) Currently (2001) under revision.

[20] ISO/IEC 16262:1998, *Information technology - ECMAScript language specification.*

[21] ISO/IEC 15145:1997, *Information technology – Programming languages – FORTH.* (Also: ANSI X3.215-1994.)

[22] ISO/IEC 1539-1:1997, *Information technology – Programming languages – Fortran - Part 1: Base language.*

[23] ISO/IEC TR 15580:1998, *Information technology – Programming languages – Fortran – Floating-point exception handling.*

[24] ISO/IEC 13816:1997, *Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP.*

[25] ISO/IEC 10514-1:1996, *Information technology – Programming languages – Part 1: Modula-2, Base Language.*

[26] ISO/IEC 10514-2:1998, *Information technology – Programming languages – Part 2: Generics Modula-2.*

[27] ISO 7185:1990, *Information technology – Programming languages – Pascal.*

[28] ISO/IEC 10206:1991, *Information technology – Programming languages – Extended Pascal.*

[29] ISO 6160:1979, *Programming languages – PL/I.* (Endorsement of ANSI X3.53-1976 (R1998) [43].)

[30] ISO/IEC 6522:1992, *Information technology – Programming languages – PL/I general-purpose subset.* (Also: ANSI X3.74-1987 (R1998).)

[31] ISO/IEC 13211-1:1995, *Information technology – Programming languages – Prolog – Part 1: General core.*

[32] ISO/IEC 8824-1:1998, *Information technology – Abstract Syntax Notation One (ASN.1) – Part 1: Specification of basic notation.*

[33] ISO 9001:1994, *Quality systems – Model for quality assurance in design, development, production, installation and servicing.*

[34] ISO/IEC 9126:1991, *Information technology – Software product evaluation – Quality characteristics and guidelines for their use.*

[35] ISO/IEC 12119:1994, *Information technology – Software packages – Quality requirements and testing.*

[36] ISO/IEC 14598-1:1999, *Information technology – Software product evaluation – Part 1: General overview.*

National and other standards documents

[37] ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*

[38] ANSI/IEEE Standard 854-1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic.*

[39] The Unicode Standard, version 3.0, 2000. Note that version 3.0 the encoded character repertoire is exactly the same as for ISO/IEC 10646-1:2000.

[40] ANSI X3.113-1987 (R1998), *Information technology – Programming languages – Full BASIC.*

[41] ANSI X3.23-1985 (R1991), *Programming languages – COBOL.*

[42] ANSI X3.226-1994, *Information Technology – Programming Language – Common Lisp.*

[43] ANSI X3.53-1976 (R1998), *Programming languages – PL/I.*

[44] ANSI/IEEE 1178-1990, *IEEE Standard for the Scheme Programming Language.*

[45] ANSI/NCITS 319-1998, *Information Technology – Programming Languages – Smalltalk.*

Books, articles, and other documents

[46] J. S. Squire (ed), *Ada Letters*, vol. XI, No. 7, ACM Press (1991).

[47] M. Abramowitz and I. Stegun (eds), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, Tenth Printing, 1972, Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

[48] J. Du Croz and M. Pont, *The Development of a Floating-Point Validation Package*, *NAG Newsletter*, No. 3, 1984.

[49] J. W. Demmel and X. Li, *Faster Numerical Algorithms via Exception Handling*, 11th International Symposium on Computer Arithmetic, Winsor, Ontario, June 29 - July 2, 1993.

[50] D. Goldberg, *What Every Computer Scientist Should Know about Floating-Point Arithmetic.* ACM Computing Surveys, Vol. 23, No. 1, March 1991.

[51] J. R. Hauser, *Handling Floating-Point Exceptions in Numeric Programs.* ACM Transactions on Programming Languages and Systems, Vol. 18, No. 2, March 1986, Pages 139-174.

[52] W. Kahan, *Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit*, Chapter 7 in *The State of the Art in Numerical Analysis* ed. by M. Powell and A. Iserles (1987) Oxford.

[53] W. Kahan, *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*, Panel Discussion of *Floating-Point Past, Present and Future*, May 23, 1995, in a series of San Francisco Bay Area Computer Historical Perspectives, sponsored by SUN Microsystems Inc.

[54] U. Kulisch and W. L. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, 1981.

[55] U. Kulisch and W. L. Miranker (eds), *A New Approach to Scientific Computation*, Academic Press, 1983.

[56] D. C. Sorenson and P. T. P. Tang, *On the Orthogonality of Eigenvectors Computed by Divide-and-Conquer Techniques*, SIAM Journal of Numerical Analysis, Vol. 28, No. 6, p. 1760, algorithm 5.3.

[57] *Floating-Point C Extensions* in Technical Report Numerical C Extensions Committee X3J11, April 1995, SC22/WG14 N403, X3J11/95-004.

[58] D. M. Gay, *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions*, AT&T Bell Laboratories, Numerical Analysis Manuscript 90-10, November 1990.

[59] M. Payne and R. Hanek, *Radian Reduction for Trigonometric Functions*, SIGNUM Newsletter, Vol. 18, January 1983.

[60] M. Payne and R. Hanek, *Degree Reduction for Trigonometric Functions*, SIGNUM Newsletter, Vol. 18, April 1983.

[61] N. L. Schryer, *A Test of a Computer's Floating-Point Unit*, Computer Science Technical Report No. 89, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

[62] G. Bohlender, W. Walter, P Kornerup, D. W. Matula, *Semantics for Exact Floating Point Operations*, IEEE Arithmetic 10, 1992.

[63] W. Walter et al., *Proposal for Accurate Floating-Point Vector Arithmetic*, Mathematics and Computers in Simulation, vol. 35, no. 4, pp. 375-382, IMACS, 1993.

[64] J. Gosling, B. Joy, G. Steele, *The Java Language Specification.*

[65] S. Peyton Jones et al., *Report on the programming language Haskell 98*, February 1999.

[66] S. Peyton Jones et al., *Standard libraries for the Haskell 98 programming language*, February 1999.

[67] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*, The MIT Press, 1997, ISBN: 0-262-63181-4.

Annex E
(informative)

Possible changes to part 1

This annex indicates possible changes to ISO/IEC 10967-1:1994 to bring it more in line with the exceptional values, terminology, etc. used in this part. A revision of part 1 is not limited to the items listed below, nor guaranteed to be done as described below.

a) Use of **infinitary** and **invalid** instead of **undefined**.

b) Use of **overflow** instead of **integer_overflow** and **floating_overflow**.

c) Use of $result_I$ (from LIA-2).

d) Use of $result_F$ from LIA-2, which implies a minor correction, plus that **overflow** and **underflow** continuation values are explicitly specified.

e) Slightly stricter parameter constraints for floating point parameters (from LIA-2).

f) Update of the rnd_error_F definition, possibly together with removal of the rnd_style_F parameter.

g) Removal of the $modulo_I$ integer parameter (replaced by new operations in part 2).

h) Removal of the div^f_I/div^t_I and $rem^f_I/rem^t_I/mod^a_I/mod^p_I$ operations, since $quot_I/ratio_I/group_I$ and $mod_I/residue_I/pad_I$ are now specified in part 2 as replacements. Note that $quot_I = div^f_I$ and $mod_I = mod^a_I = rem^f_I$.

i) Specify the LIA-1 operations also for IEC 60559 special values as arguments, like in LIA-2, including using the no_result_F and $no\_result2_F$ helper functions.

j) Change "data type" to "datatype".

k) Change "denormal" to "subnormal" for consistency with IEEE 854 and C99.

ICS 35.060

Price based on 174 pages